Implementation of Data Compression and FFT on TinyOS

Ning Xu
Embedded Networks Laboratory
Computer Science Dept. USC. Los Angeles
email: nxu@usc.edu

Abstract

In this paper we describe implementation issues, challenges and design trade-offs in developing the compression and FFT components in TinyOS. We choose to implement Lempel-Ziv algorithm and Sorensen’s algorithm in these 2 components respectively. The code has been tested in Mica mote, both components work well under certain circumstances (sampling mode, sampling rate etc.). It is our hope that the code and some lessons learnt along the way can be helpful to other TinyOS developers.

1 Introduction

In sensor network applications such as structural health monitoring, the data volume is potentially huge. While sensor nodes have limited memory and radio communication is power-hungry, data reduction methods should be applied to the dataset before it is stored or transmitted. We have implemented data compression and FFT components in TinyOS, this paper discusses the implementation issues, performance and limitations of the components.

One question to ask is: why should we do compression and FFT on TinyOS? The benefits of compression in data reduction are obvious, especially when the samples exhibit heavy repetition pattern. The idea of FFT as a data reduction method is not as straightforward. FFT can reduce dataset in 2 ways: first, a smaller dataset is needed to represent the signal at an acceptable level in frequency domain. For example, after a N-point FFT we can pick \( \frac{N}{2} \) interleaving data points to represent the signal, assuming that a resolution of \( 2f/N \) is sufficient, where \( f \) is the sampling rate. By doing this we reduce the dataset size by a factor of 2; second, \textit{ANPSDs} (Average Normalized Power Spectral Densities) can be used to significantly reduce the size of dataset while preserving system identification information in some applications. For example, the natural frequencies of a bridge are determined as the peaks of \textit{ANPSDs} in response to ambient vibration[6]. When computing \textit{ANPSDs}, firstly a "Window-length" of \( N \) data points is taken, FFT is then performed to get the PSD, subsequently, the PSDs are computed for all succeeding blocks of \( N \) data points, finally the PSDs are averaged, and a peak detection algorithm can be run to compute natural frequencies. If \( m \) blocks of data are taken to compute the \textit{ANPSDs}, as a result, the size of the dataset is reduced from \( m \cdot N \) to \( N \), by a factor of \( m \).

2 Design challenges and algorithm selection

TinyOS is an event-driven embedded operating system that is specifically designed for wireless sensor network in UC Berkeley. The three computational abstractions of TinyOS (\textit{i.e.}, commands, events and tasks) should be as small as possible, so that high concurrency can be achieved at a fine granularity[1]. At the same time, other system constraints exist on the hardware side. For example,
Mica mote comes with 4Mhz processor, which makes it a bottleneck in computation-intensive components such as compression; no floating point unit, which significantly slows down FFT computation; a 4KB RAM, which requires careful plan in memory allocation; and a notoriously low speed EEPROM, which suggests that additional attention must be paid to schedule the logger writing at a suitable rate.

Although the compactness of TinyOS is unarguably desirable, the lack of primitives in TinyOS to handle concurrency unfortunately puts real-time applications at processor’s mercy: if the processor is not fast enough, problems come up. In our design, there exist race conditions among sampling, compressing/FFT and logger writing. Thus good buffering design is key to the correct functioning. We have adopted a simple double-buffering scheme for sampling and processing in our components, the detail will be discussed later in the implementation section.

Given the above system constraints, mainly the processor speed, memory size and concurrency requirements, we identify 2 criteria in algorithm selection: first, we should choose algorithms that have low space&time complexity; second, the decomposability of the algorithm, i.e., complex task should be able to be decomposed into serials of smaller tasks.

We next present a brief overview of the existing Compression/FFT algorithms, explain why we choose to implement Lempel-Ziv algorithm and Sorensen’s algorithm.

2.1 Data compression

The aim of data compression is to reduce redundancy in stored or communicated data. The data compression methods can be classified along the axis of lossless and lossy, static and dynamic. While a complete treatment of the discipline of data compression is outside the scope of this paper, we intend to survey some classic methods and identify potential candidates. Classic lossless compression methods include Huffman coding, Running-length Coding and Arithmetic coding. Modern compression methods go back to J. Ziv and A. Lempel, who published 2 seminal papers in 1977 and 1978 respectively. Many of the commonly used UNIX compression utilities (gzip, compress etc.) are based on LZ77/LZ78. In general, the dictionary-based LZ78[5] is an improvement over sliding Window based LZ77[4], but the former is much more memory-intensive since the algorithm needs to build and maintain a dynamic dictionary while compressing. We identify LZ77 as a candidate because it is an in-place algorithm, and can be decomposed to smaller tasks in TinyOS implementation. Our implementation adopts LZSS\(^2\), a variant of LZ77.

2.2 FFT

Fast Fourier Transform (FFT) is a fast algorithm to perform frequency analysis of discrete signal. The classic FFT algorithm takes \(N\) (usually a power of 2, otherwise padding is necessary) complex values as input, reshuffles the input array by bit reversing, computes the result by employing a divide-and-conquer strategy. The classic FFT is memory inefficient when dealing with real value input, since it wastes memory to store the imaginary part which is initially set to zero. To solve this problem, many real-value FFT algorithms, such as Sorensen’s algorithm, Bruun-FFT and Packing algorithm, are proposed. These algorithms can reduce the memory requirement by a factor of two by exploiting the symmetry properties of the Fourier-transform. We have chosen to implement split-radix algorithm of Sorensen because it can be implemented in-place and it is the fastest according to [3].

\(^1\)In our experiment, the maximum logger writing speed is 32 lines per second, higher writing rate results in loss of logger line.
\(^2\)LZSS outputs a 2-tuplet, while standard LZ77 outputs a 3-tuplet.
3 Implementation issues

The code is written in NesC. It is tested with Mica mote. The sampling process has 2 different modes: continuous mode and timer-driven mode. In continuous mode, the sampling is driven by the clock of the ADC itself, the highest sampling rate can go to 16KHz. In timer-driven mode, the timer is set to fire at a given rate, at each fire a command is called to read the desired ADC channel. The timer-driven mode is suitable for real-time continuous monitoring, in which sampling, compressing/FFT and logging are interleaved.

3.1 Lempel-Ziv coding (LZ77)

The pseudo code of the compression task is as follows.

```
task void lz77() {
    if ( lookAheadBuffer not empty) {
        search the longest match in the sliding window
        for the lookahead buffer;
        if( length > MINIMUM_MATCH_LENGTH ) {
            output a (length, position ) pair;
            shift the sliding window length samples along;
        } else {
            output the first sample in the lookahead buffer;
            shift the sliding window 1  samples along;
        }
    }
    post lz77();
}
```

We now describe the implementation in more details.

- **Task decomposition**
  Basically, the LZ77 algorithm contains nested loops: outer loop scans through the input array, the inner loop searches the longest match in the sliding window for the look-ahead buffer, when the longest matched pattern is found, the algorithm outputs a \((a, b)\) pair as codeword, where \(a\) is the length of the matching pattern, \(b\) is the relative pointer to the start of the matching pattern. The compression task is decomposed into serials of smaller tasks, each of which is equivalent to an inner loop, at the end of each task it posts itself, which is equivalent the outer loop, to process next trunk of data.

- **Double buffers for sampling and logger writing**
  Double buffers are used in the timer-driven mode. When a buffer fills up, a processing task will be posted, the new incoming samples will go to the other buffer. The sampling rate must not be higher than the processing (compressing or FFT) rate, otherwise the sampling buffer will be overridden. Currently there are no built-in mechanisms (flow control, for example) to avoid the buffer overriding situation, but rather lights up the red LED as an indicator should such a situation happens. It should be pointed out that most of the compression algorithms are extremely susceptible to errors, in our case it is impossible to restore the data if sampling buffer gets overridden.

- **Logger issue**
  Recognizing the logger works at a maximum rate of 32 lines per second, we use a boolean variable, LogClear, to control the logger writing rate. After Logger write command is issued, LogClear is set to FALSE, it will be set to TRUE after 1/32 second. Whenever a task detects
Table 1: Compression ratio of actual experiment runs

<table>
<thead>
<tr>
<th>Experiment Description</th>
<th>Original Data length</th>
<th>Code Data Length</th>
<th>Compression Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quiescent</td>
<td>800</td>
<td>66</td>
<td>12.1</td>
</tr>
<tr>
<td>Tapping the table</td>
<td>800</td>
<td>230</td>
<td>3.44</td>
</tr>
<tr>
<td>Mote in mildly shaking hand</td>
<td>800</td>
<td>242</td>
<td>3.30</td>
</tr>
<tr>
<td>Mote in strongly shaking hand</td>
<td>800</td>
<td>328</td>
<td>2.44</td>
</tr>
</tbody>
</table>

that the LogClear is FALSE, it posts itself and exits. This ensures that the logger writes correctly at maximum rate.

- **Parameter adjustment**

  The adjustable parameters include buffer size, sliding window size and maximum matching length. In Mica mote with 4KB RAM, the maximum buffer size is around 1700 bytes (850 samples). The algorithm is quite computationally expensive in that lots of the computations go to searching the longest match. How to choose the sliding window size and maximum matching length is a trade-off between algorithm speed and compression ratio.

### 3.2 Sorensen’s algorithm

Sorensen’s algorithm[2] works by omitting the unnecessary operations in the case of real input, exploiting the symmetry properties of the Fourier-transform. The algorithm was implemented in [2], we simply port it to NesC as a TinyOS task. It takes as input the real array \( re(0), re(1), \ldots, re(N-1) \), produces the output as \( re(0), re(1), \ldots, re(N/2), im(N/2-1), \ldots, im(1) \). It is an in-place algorithm, and requires \( O(n\log(n)) \) floating point multiplications and additions. The problem here is the ATMEL Mega128 micro-controller used in Mica mote does not support floating-point operation directly, floating operation is done by emulation, which is slower by orders of magnitude. As we have observed in the experiment, the FFT component takes more than 30 seconds to complete a 512-point FFT, while the same computation will only take hundreds of millisecond on a Pentium-4 machine running Matlab.

### 4 Current status

The compression component works well in both continuous sampling mode and timer-driven sampling mode. In continuous sampling mode, samples can be taken at highest rate of 16KHz until the buffer is filled up, then the compression task is posted to process the data. In timer-driven sampling mode the compression component works in highest sampling rate of 128Hz, higher sampling rate will result in loss of data.\(^3\)

Our experiment shows the compression ratio is quite satisfactory, as shown in Table 1. The size of dataset can be reduced by an average factor of 3, if there is no apparent vibration, the compression ration can go to as high as 12. In those experiment runs, parameters such as buffer size, sliding window size and maximum matching length are set to 800, 100 and 16 respectively, the sampling rate is 128Hz.

The FFT component is not very useful in timer-driven sampling mode since FFT is so slow that it can not keep up the pace with a reasonable sampling rate (e.g., 16Hz). It is shown that the Mica mote takes more than 30 seconds to complete a 512-point FFT, this fact suggests that we

---

\(^3\)Tested in Mica mote, with sliding window size of 100 and max matching length of 16. The sampling rate can go up a bit if those 2 parameters are decreased.
should not run FFT in timer-driven sampling mode. The component works well in continuous sampling mode. As we have argued in the introduction section, it can be used to reduce dataset by computing the **ANPSDs**.

The current code supports single channel compressing/FFT, in consideration of the constrained system resource. Although it is straightforward to extend it to compress/FFT data from multi-channel, new design challenges come up in multi-channel sampling. For example, in accelerometer sampling, if samples from x/y axes are combined in one buffer and compressed, the compression ratio is expected to drop significantly due to less repetitions, however, if samples are compressed in separate buffers, it would require doubling the memory size to achieve the same compression ratio as in single channel situation.

Last but not least, although the component is designed to compress/FFT the accelerometer samples, it is fairly ease to modify the component to process samples from other sensors (e.g., magnetometer, acoustic, light etc.) on Mica mote sensorboard. All needed to be done is to modify the wiring in the configuration file.

## 5 Future work

We are considering some improvements in the future.

- **Fixed-point library support in TinyOS**

  Fixed-point math is a system by which integer variables can represent fractional values with a fixed precision behind the decimal point. Fixed-point math is much faster to execute than floating-point math, and should be used when a limited amount of fractional precision is sufficient. We expect that once the fixed-point library is in place, the speed of FFT can be improved dramatically.

- **Fast Logger**

  The current Logger component (eepromM.nc) writes data to EEPROM in the unit of line, there is a 5ms pause between each line writing, which slows down the speed. Some improvements can be make to speed up the logger, such as writing the EEPROM in the unit of page, or increasing the size of logger line (currently it is set to a compromising value of 16).

- **Hardware implementation of data pre-processing**

  It might be worthwhile to look at the possibility of implementing pre-processing methods, such as data compression, FFT and filtering etc., in the future Mica version. Another question to ask is, can we add a floating-point co-processor to improve the sensor node’s local computing capability?

## References


