Using Data Compression for Energy-Efficient Reprogramming of Wireless Sensor Networks

Nicolas Tsiftes <nvt@sics.se>

Swedish Institute of Computer Science
Box 1263, SE-164 29 Kista, Sweden

SICS Technical Report T2007:13
ISSN 1100-3154
ISRN:SICS-T–2007/13-SE

December 18, 2007

Keywords: wireless sensor networks, data compression, reprogramming, experimental evaluation, energy efficiency

Abstract

Software used in sensor networks to perform tasks such as sensing, network routing, and operating system services can be subject to changes or replacements during the long lifetimes of many sensor networks. The task of reprogramming the network nodes consumes a significant amount of energy and increases the network congestion because the software is sent over radio in an epidemic manner to all the nodes.

In this thesis, I show that the use of data compression of dynamically linkable modules makes the reprogramming more energy-efficient and reduces the time needed for the software to propagate. The cost for using data compression is that the decompression algorithm must execute for a relatively long time, and that the software takes five to ten kilobytes to store on the sensor nodes.

Contents

Chapter 1

Introduction

Wireless sensor networks are today used for many different applications, such as environmental monitoring, object tracking, intrusion detection, and industrial processes. The networks consist of embedded devices that are equipped with sensors and radio transceivers. We refer to these embedded devices as *sensor nodes*. The sensor nodes collaborate to sense their surrounding environment and communicate their results to one or more data sinks. The concept of wireless sensor networks introduce several challenges for researchers and programmers because the nodes have limited energy supplied by batteries, and very constrained computational resources in order to keep the monetary cost low.

Before deployment, the sensor nodes are loaded with software that has been installed after receiving the software from a computer connected by wire. It is however possible that bugs are later found in the software, or that new software must be installed. The software should preferably be propagated over-the-air since it would be a time-consuming and tedious task to collect the nodes and reprogram them by wire. Also, the nodes could be performing an operation that should not be disrupted for a longer period, or they could be located in a remote position.

This technical report shows that the use of data compression reduces the energy cost of distributing and installing dynamically linkable modules in wireless sensor networks.

1.1 Problem Statement

One significant design issue in wireless sensor networks is to preserve energy in order to extend the lifetime of the sensor nodes. The radio transceiver is one of the most energy-consuming devices on sensor nodes and this makes it important to put the radio to sleep as much as possible. Because of the flooding nature of software propagation protocols, the distribution of software can cause a considerable increase in the energy consumption of the sensor nodes. Furthermore, low data rates and high packet loss rates can lead to long propagation delays for the software that are magnified proportionally to the size of the software image.

On the sensor boards used in this report, the radio transceiver consumes a current that is an order of magnitude higher than the active micro-controller. It is therefore interesting to explore the trade-off between the reduction of the radio transmission energy that is achieved by using data compression, and the energy that is required for executing the decompression software at the sensor nodes. Another important issue

is to reduce the radio traffic that is required for the reprogramming so that the sensor network can do its ordinary tasks with minimal disruption.

Much of the intellectual effort of the report lies in the implementation of existing data compression algorithms. The main difficulty is to adapt memory-demanding algorithms to a target platform that has very constrained memory resources. Also, there is no memory protection and the debugging possibilities are limited.

1.2 Method

The method I use in this technical report is that of experimental computer science. I implement a number of data compression applications and then I run and measure their energy-efficiency and performance. Initially, I have studied the field of reprogramming wireless sensor networks and data compression to determine which algorithms may be feasible to implement. Thereafter, I have implemented the decompression software in Contiki and integrated it with the code propagation application. In several of the cases, I have also implemented corresponding compression software. The results are quantified in terms of energy consumption, compression ratio, memory requirements, code size, and execution speed. Moreover, the trade-offs between algorithm computations and radio traffic reductions are analyzed.

1.3 Limitations

Due to the large number of variations of data compression algorithms, only a limited set was chosen from the main types of algorithms. The selections were made based on which algorithms are deemed feasible to implement on sensor nodes. The variations are however often small, so the characteristics of the main types of algorithms should be represented well by the chosen variations.

Given enough time, further optimizations could improve the energy-efficiency of certain implementations, as the decompression time has a strong influence on the energy consumption.

1.4 Alternative Approaches

Reprogramming of wireless sensor networks is a research topic that has a number of different challenges. First, it is a question of reliably distributing the software to all the nodes in the network with a minimal use of radio traffic. Second, it is a question of how the software is represented in memory and persistent storage. The initial approach has been to use full-image replacements that are broadcasted over radio [18]. After the complete image has been received, the old image is over-written with the new image in an external storage such as EEPROM or flash memory. In response to this inefficient technique, which requires much redundant data to be sent, there have been several different suggestions to make the reprogramming phase more energy-efficient and to improve the propagation speed.

Among the alternative solutions related to this report, difference techniques have been shown to be very energy-efficient when an earlier version exists in the sensor network, and the changes are small in relation to the size of the previous version. Instead of sending the full system image, only the difference between the latest version and a previous version is sent. However, this technique requires that either all the

sensor nodes in the network run identical copies of an old version of the software, or that a version control system is in place. Other approaches have been to use scripting languages or virtual machines for more compact software, but these approaches require a significant execution overhead once the software is installed and running. These techniques are discussed in more detail in section 2.1.1.

1.5 Scientific Contributions

This report contains two scientific contributions. First, I show that it is possible to implement and use several decompression algorithms, including the popular GZIP, on sensor nodes with very constrained processing and memory resources. Second, I evaluate the implementations and show that the use of data compression on dynamically linkable modules is an energy-efficient technique for reprogramming wireless sensor networks over the air.

1.6 Report Structure

The remaining part of this report is structured as follows. In chapter 2, I give an introduction to wireless sensor networks and their special research problems. After this, I introduce the reader to the Contiki operating system, in which the implementations are made. The background part of the report is ended with an overview of the relevant compression algorithms that I have implemented. The design and implementation choices are then covered in chapter 3, followed by the evaluation in chapter 4. The report is concluded in chapter 6, along with a discussion of the results and suggestions for future work.

Chapter 2

Background

This chapter introduces the reader to the two fields that this report relates to, namely wireless sensor networks and data compression. In the section on wireless sensor networks, I also describe the Contiki operating system and how it is adapted to memory-constrained sensor nodes. The data compression implementations for Contiki that are described in chapter 3 use the algorithms that are introduced in section 2.2.

2.1 Wireless Sensor Networks

Wireless sensor networks [1] consist of small sensor nodes that collaborate to gather data from the physical conditions of their surrounding environment. Sensor nodes can form potentially large networks ranging from a few to thousands of nodes in challenging conditions.

The novel possibilities and challenges of wireless sensor networks have received considerable interest from the research community during the latest years. Sensor networks impose two additional constraints on software and protocol design problems: very limited energy and computing resources.

The general characteristics of sensor nodes are that they are small, very resource-constrained and have a relatively low monetary cost. This makes wireless sensor networks practical for covering large areas and to be deployed in harsh environments where it would not be feasible to use wired nodes. Typical uses include building automation systems, disaster discovery, environmental and wildlife studies, mobile entertainment, process monitoring, and surveillance.

Sensor nodes have been designed with modest hardware resources in order to keep the monetary cost low and to require very little energy. Their equipment usually consists of one or more sensors, a small micro-controller and a wireless communication device. Energy is normally supplied by on-board batteries, but solar cells have also been used [32].

In many types of applications, the duration of the deployment is expected to be long. Individual sensor nodes might be inaccessible during this period, so battery changes are not practical. All nodes in the network must therefore operate efficiently in terms of energy so that the whole network can function as long as required.

2.1.1 Software Updates in Wireless Sensor Networks

Software updates in sensor networks are required for a multitude of different reasons. Dunkels et al. [10] discuss the following possible scenarios: software development,

sensor network testbeds, correction of software bugs, application reconfiguration and dynamic applications. After the sensor nodes have been deployed, they are not likely to be easily accessible for being reprogrammed by wire. Instead, software updates must be sent over-the-air. The radio transmission can cause a considerable energy usage, because the current consumption of the radio transceivers examined in this report are an order of magnitude higher than the micro-controller being in active mode. This makes it important to have an energy-efficient reprogramming technique that reduces the costly radio traffic. Several methods exist for reprogramming a sensor network:

Full image replacement With this approach, a complete system image is built and statically linked when a software update must be made. This scheme is inefficient because a large amount of data that already exist at the sensor nodes must be transmitted over-the-air and re-written to program memory. Another downside of this method is the significant delay in distributing the image to all the nodes in the network if the data rate is low.

Incremental updates A delta (or difference) from a previous version of the software is generated at the base station. The delta is then transmitted out to all the nodes in the sensor network. It is then applied individually on each sensor node and the old module is replaced in memory with a new version. Sending the delta instead of the complete module can reduce the size of the transmitted data significantly [17, 21, 25, 28]. Incremental linking is a scheme that can improve the efficiency of delta generation by using some knowledge of the executable format.

Incremental updates can pose a problem in heterogeneous networks where all nodes do not necessarily run the same version. In such environments, a version control system is required both at the base station and at the sensor nodes. Moreover, if a major change or a new module must be distributed to the nodes, then the delta size will be dependent on the compression technique in use.

Virtual Machines The idea behind this approach is that virtual machine code can be made smaller than corresponding native code. Common high-level abstractions can be expressed by a few VM byte codes. The reduction in the size of the code that is transfered over the network will give energy savings up to a certain break-even point, where the overhead of running the code under the virtual machine for a long time offsets the energy savings. An example of a VM is Maté [23], which is a stack-based virtual machine architecture developed for sensor networks.

Dynamically linkable modules Operating systems such as Contiki [10] and SOS [13] support run-time dynamic linking of modules. Contiki provides an ELF loader [6] so that modules can be built with a popular cross compiler such as GCC and later loaded into the system without intermediary processing. The difference between Contiki and SOS is that Contiki does run-time relocation and SOS uses position-independent code. The aim of this report is to further improve the efficiency of this approach by using data compression.

2.1.2 The Contiki Operating System

Contiki [8, 11] is a lightweight open-source operating system that is specifically designed for resource-constrained computer platforms such as sensor nodes. Included in Contiki is a kernel, a very small TCP/IP implementation [9], a set of hardware drivers, and a variety of application modules. The hardware-independent part of Contiki is written in the C programming language. The system is designed to be portable and a number of different platforms are supported, for example the Scatterweb Embedded Sensor Board [2] and the Moteiv Tmote Sky board [26].

The system design is based on an event-driven execution model. This design is often used in operating systems designed for resource-constrained environments because the system can provide concurrency among programs using much less memory than a thread-based system.

All programs share a single stack together with the kernel. This leads to a more effective use of the memory resources in sensor nodes that do not have a virtual memory mechanism. If each process would have its own stack, then its size would have to be guessed beforehand and often over-allocated by a large margin in order to be on the safe side. Another advantage with the event-driven system design is that there is no need for locking mechanisms to protect shared variables.

Run-Time Dynamic Linking of ELF Modules

Contiki provides support for dynamic linking and loading of modules in the standard Executable and Linking Format [6]. ELF files contain sections of executable code, data, symbols and other information necessary for linking and relocation. figure 2.1 shows how an ELF file can be structured. A motive for choosing ELF is that it has gained a widespread use and a number of tools exist for generating and manipulating ELF files. Since the format is designed for 32-bit architectures, and the sensor nodes are typically run on 16- and 8-bit micro-controllers, there can be a significant part of redundant information in the file. Contiki addresses this problem by also providing a modification for ELF called Compact ELF (CELF), that uses 16-bit data fields where it is possible [10]. CELF operates most efficiently when the relocation sections are relatively large compared with other sections, since CELF does not transform instructions code, for example.

The ELF sections contain information of different characteristics. For example, the data section often contains large quantities of zeroes and the symbol table will usually include much English text. Such information known a priori could potentially be used by a compressor to further improve the compression ratio.

Application Development in Contiki

Event-driven applications are usually implemented as state machines. Software designs that are naturally described with logically blocking sequences are not easily adapted to a state machine design and they become difficult to understand and maintain. Contiki provides a very small thread library called Protothreads [7, 12] to simplify the design and implementation event-driven software. Programs that use Protothreads can be implemented with conditional blocking wait operations.

Protothreads are stack-less and require cooperation from each thread. They are not preempted by the Contiki kernel and must therefore return control to the scheduler when they find it suitable in order to make the system responsive. An example

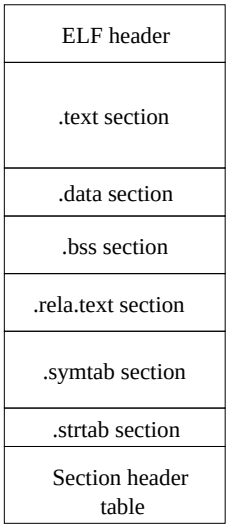


Figure 2.1: A typical ELF layout for Contiki software.

of such a long computation that is relevant to this report is a decompression process. Such a process could potentially monopolize the use of the processor for over ten seconds if it does not cooperate. Time measurements of the implemented decompressors are available in section 4.6.

Choosing the right time to yield can be difficult, but a more important problem is that rewriting the software to have this capability can make it more complicated if the function call tree is several levels deep. To allow programs to stay in their original form while still cooperating, Contiki includes an additional preemptive Multi-Threading library (MT), which may be linked with an application. By using this library, applications can call a yield function in MT that transfers control back to the scheduler. These calls can be placed where the programmer finds it suitable. Automatic variables that are allocated on the stack are not saved between scheduled executions of a process, so the process state is typically held in static variables instead.

A Contiki process is defined by an event-handler at the top level. Processes communicate with one another by posting events. Events can optionally be broadcasted to all processes. Two types of events exist: synchronous events are processed immediately by the receiver, while asynchronous events are queued until the receiver is scheduled to run. A pointer to an opaque data area may optionally be sent along with the event. Optionally, it is possible to use a service mechanism that enables processes to register functions that can be called by other processes directly.

2.2 Data Compression Algorithms

General-purpose compression algorithms are normally designed with a high compression ratio as the primary goal. We are interested in compression techniques that primarily make software distribution and installation in sensor networks more energy efficient. In order to help us to understand the implementation issues and traits of the algorithms, this chapter gives an overview of the existing data compression techniques that we focus on and evaluate in this report.

2.2.1 Lossless and Lossy Algorithms

Compression algorithms can be either lossy or lossless. Lossy algorithms can give substantially better compression ratios in some cases because they perform non-reversible changes. One scenario in which this is acceptable could be digital sound compression. Lossless algorithms restore the original data perfectly. A general-purpose algorithm which can not depend on any knowledge on the input data must be lossless. This is obviously necessary when compressing software, because one single bit of error can lead to a disastrous run-time failure.

Lossy methods are viable in some circumstances when compressing software. The requirement in such a case is that the compressed program operates exactly the same as if it had not been compressed. An example of a lossy compression that is possible for software is a compiler that optimizes for space by transforming certain operations into equivalent operations that use less storage, or removes unused functions.

2.2.2 Statistical Algorithms

Statistical algorithms consist of two parts, namely statistical modeling and coding. Compression ratios will largely depend on how accurately the statistical modeling is done. Statistical modeling can be done in the following ways:

Fixed The model is predetermined and agreed on by the encoder and decoder.

Semi-adaptive One pass is made over the data to construct the model.

Adaptive The model is constructed on the run.

For the decoder to function properly, the model has to either be transmitted with the sent message or be an adaptive model, which is built on-the-run by the encoder and the decoder. The goal of these algorithms is to minimize the redundancy found in the message by the statistical modeler. A model of order N takes N previous symbols into account. In general, better results are obtained by increasing the order of the model. The problem is that models of a higher order than zero usually require much more memory than what is available in sensor nodes.

Once the modeling is finished, the encoder will optimally compress symbols with an average code length close to the information entropy. The order-0 information entropy of a random variable X is defined as

$$H(X) = \left[-\sum_{i=1}^{n} p_i \log_2 p_i \right],$$

where n is the number of symbols in the alphabet, and p_i is the probability of the ith symbol in the alphabet. The information entropy is a bound for how effective the encoder can be.

Huffman Coding

Huffman Coding [15] is a greedy algorithm for constructing minimum-redundancy codes for symbols that exist in a message. These codes can be of variable-length where symbols with a higher probability of occurrence are assigned shorter codes. Each code has a unique bit pattern prefix so that the Huffman encoded data can be decoded unambiguously.

```
begin
    Input: F: a forest of trees. Each tree will initially have one node, which
           consists of a unique symbol and its weight.
    Output: T: A Huffman trie.
    while |F| > 1 do
        T_1 ← FindMinimum (F);
        Remove (F, T_1);
        T_2 ← FindMinimum (F);
        Remove (F, T_2);
        T_new ← CreateTree ();
        SetLeft (T_new, T_2);
        SetRight (T_new, T_1);
        SetWeight (T_new, Weight (T_1) + Weight (T_2));
        Insert (F, T_new);
    end
    return F;
end
```

Algorithm 1: *Huffman code generation.*

We follow the outline of Huffman Coding as shown in algorithm 1. Huffman's algorithm operates on a forest of trees. Initially, each symbol is represented by a distinct tree and its weight corresponds to the symbol's probability. In every iteration, the two trees with the lowest weights are removed from the forest. A new tree is formed with these two trees as leafs and a weight that is the sum of the weights of the leafs. The new tree is then added to the forest. When only one tree remains, the algorithm stops and this tree becomes the Huffman tree. The weight of the final tree is the sum of the symbol weights. Code lengths are assigned to all symbols corresponding to their depth in tree. For instance, consider that we want to encode the string S = "ABAACAACEDDD" with Huffman's method. We determine the probability of the symbols to be $A : \frac{5}{12}$, $B : \frac{1}{12}$, $C : \frac{2}{12}$, $D : \frac{3}{12}$, and $E : \frac{1}{12}$. The constructed Huffman tree is shown in figure 2.2. By following the path to each leaf, we obtain the binary codes for the symbols that are $A = 0$, $B = 1110$, $C = 110$, $D = 10$, and $E = 1111$. Thus the string S is encoded in binary as $H(S) = 011100011000111011111101010$. The length of $H(S)$ is 25 bits, compared with the 96 bits required for S.

The relative overhead of sending the Huffman codes along with the compressed message grows in inverse proportion to the size of the message. When compressing small applications for sensors, this can be a significant cost. However, this can be done more efficiently by using canonical Huffman codes. This technique uses only

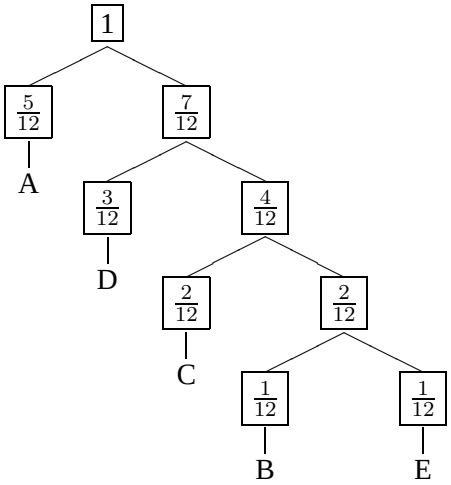


Figure 2.2: Example of a Huffman tree with the weights enclosed in the boxes.

the code lengths and a set of rules for ordering symbols with the same code lengths. It is possible to reconstruct the codes with this information only.

Adaptive Huffman coding can be used to build and update the codes on the run, so it no longer becomes necessary to transmit the statistical model with the data. One can also do higher order modeling without having to send a large model along with the compressed data.

Arithmetic Coding

Arithmetic Coding can be considered as a generalization of Huffman Coding. Huffman coding is only optimal when the symbols have probabilities of 2^{-n} because each symbol is assigned a code of an integral number of bits. One example of this is a two-symbol alphabet A, B where A has probability 0.9 and B has probability 0.1. The Huffman algorithm assigns codes with the same lengths to the symbols.

```
Input: M: the message to be encoded
Output: C: the encoded message
begin
    low ← 0;
    high ← 1;
    while MoreSymbols (M) do
        ch ← NextSymbol (M);
        interval ← high − low ;
        low ← interval × GetLow (ch);
        high ← interval × GetHigh (ch);
    end
    return low;
end
```

Algorithm 2: *Arithmetic Encoder.*

As shown in algorithm 2, Arithmetic Coding assigns a number in the range $[0, 1]$ for the entire file. The algorithm starts with doing a pass over the data to calculate the frequency of each symbol. The symbols then get non-overlapping subintervals of the interval $[0, 1)$ to start with. The intervals are half-open and their sizes are proportional to the probability of the symbol. On the downside is that Arithmetic

Coding is usually significantly slower than Huffman Coding [14]. The reason for this is Arithmetic Coding typically requires several multiplications for each processed symbol.

Next, we show an example of arithmetic coding using string "ABCCC" to be compressed. The three-symbol alphabet and the corresponding probabilities and intervals is shown in table 2.1.

Table 2.1: Probabilities and intervals for the alphabet.

Symbol	Probability	Interval
A	0.2	$[0.0 - 0.2)$
B	0.2	$[0.2 - 0.4)$
C	0.6	$[0.4 - 1)$

Table 2.2: A practical example of arithmetic encoding. The output is the low value of the last interval, that is 0.27136.

Next Symbol	Current Interval
A	$[0 - 0.2)$
B	$[0.24 - 0.28)$
C	$[0.256 - 0.28)$
C	$[0.2656 - 0.28)$
C	$[\underline{0.27136} - 0.28)$

As shown in table 2.2, when a symbol is encoded the interval becomes narrower. More decimals are therefore required to describe the low and $high$ values. Arithmetic Coding is effective because symbols with a high probability make the number of decimals in the interval grow at a slower pace, thus requiring less space to describe the values.

Run-Length Encoding

Run-Length Encoding (RLE) is a very simple lossless compression method. The encoding algorithm replaces N consecutive symbols of S with a pair (S, N). This pair must be distinctive from ordinary input for the decoder. One method is to use an escape character that seldom or never occurs in the input data.

2.2.3 Ziv-Lempel Algorithms

Ziv and Lempel published a ground-breaking paper in 1977 that described a sliding window method for compression, later called LZ77 [33]. This technique takes advantage of using references to previously encountered strings that are repeated. The fundamental difference compared with statistical algorithms is that this algorithm encodes a variable-length string with a single token, while a statistical algorithm encodes a single symbol with a variable-length code. LZ77 and several of the variants based on it are asymmetric in the sense that decompression is a much faster operation than compression.

An LZ77 compressor encodes a reference to previously seen data with a a ($length$, $distance$, $symbol$) token. It denotes the length of the duplicated data, the backwards

distance to where it starts and the symbol in the data. A single symbol S that has not been seen before is encoded as $(0, 0, S)$.

The window is divided into a history buffer and a look-ahead buffer. The history buffer is typically several kilobytes large and the look-ahead buffer is less than 100 bytes long. One important feature is that it is possible for the length to be larger than the distance.

A year after publishing their first paper on the subject, Ziv and Lempel proposed a dictionary algorithm for compression. This algorithm, later called LZ78, builds a dictionary of previously seen strings instead of using a sliding window. The length does not have to be included in the code since it is stored separately in the dictionary. Theoretically, the size of the dictionary can grow arbitrarily, but must in practice be restricted.

Several modifications of Ziv and Lempel's algorithms have been proposed. LZSS is an algorithm that offers three improvements over LZ77. The most important of these is that LZSS reference tokens are of the form $(distance, length)$. Each symbol is instead prepended with a flag that states whether the symbol is encoded or not. The Lempel-Ziv-Welch (LZW) algorithm is a modification of LZ78 which does not use character codes in the tokens, so only a dictionary reference is required.

Two specific methods based on the Ziv-Lempel sliding-window algorithm are implemented in this report. A brief description of each follows.

DEFLATE

The DEFLATE algorithm divides the compressed data in variable-length blocks and selects between three methods to use on each block individually. Figure 2.3 shows a flowchart of how the a DEFLATE decompressor operates on each block.

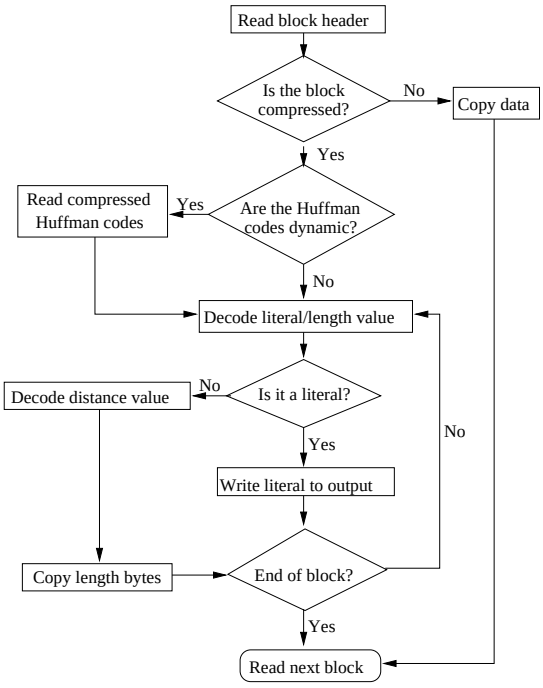


Figure 2.3: Schematic view over the decompression of a DEFLATE block.

The selection of the method depends on the characteristics of the input data. The methods are as follows: *compressed* blocks use a technique that combines slid-

ing window compression with Huffman coding. There is one set of codes each for literals and ($distance$, $length$) pairs. The Huffman codes prepend the block and are represented in canonical format, i.e., only the code lengths for each symbol are specified. A set of rules are then applied to construct the Huffman codes from the canonical codes. *Fixed* codes is a less common method, in which the codes are predefined. This is essentially the same method as the compressed, except that no statistical model is needed to store in the block. The third method is *uncompressed* which is useful for blocks that contain very few statistical redundancies, which makes it difficult to compress the data.

Difference Compression for the VCDIFF Format

VCDIFF [20] is a generic format for compactly storing the difference between a source file and a target file. This relates to ordinary data compression because VCDIFF uses a technique similar to sliding-window compression. The window is instead of a fixed size and it is prefilled with the source file. Files that are larger than the maximum window size are divided into several windows.

Four instructions are defined to describe the differences. *ADD* puts new data from the current offset into the the target file. *COPY* finds data that is references either in the old file, or in the target file and copies it to the end of the target file. *RUN* adds a capability for run-length encoding. It copies a byte from the data section a number of times into the target data. An conceptual view of this is shown in figure 2.4.

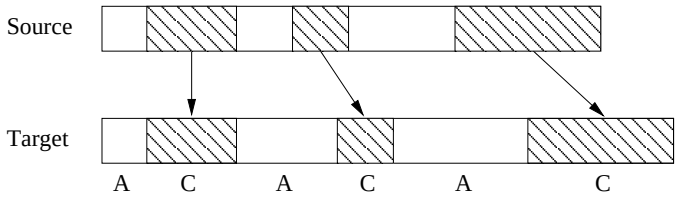


Figure 2.4: A conceptual view of how differences are described. A *denotes the ADD instruction, while* C *represents the COPY instruction.*

A VCDIFF file contains a number of windows, each having sections for data, instructions and addresses. The decoder maintains a current offset for each of these sections. VCDIFF encoders can also compress files without the referencing to another source file. In this case, the target is itself regarded as the source. The compression is however not as efficient as for example DEFLATE, because it does not use a statistical compression method. Hence, it is possible to compress the deltas further by using a secondary compressor.

2.2.4 Transformation Algorithms

The purpose of a transformation algorithm is to re-arrange the data so that it becomes better suitable for compression by another method. The transformation stage itself does not compress the data, but can actually expand it by a small fraction.

Move-To-Front Coding

Move-To-Front Coding (MTF) is a transformation algorithm that performs well if there is a high degree of locality of reference in the data. MTF will then assign lower

values to symbols that occur often in a context. MTF is a fast algorithm because only one pass needs to be made over the data.

MTF encoders and decoders store an alphabet of symbols in a list that is sorted in an order based on recency. The next symbol that is encountered in the data is searched for sequentially in this list. The position of the symbol is written to the output and the symbol is then moved to the front. After the transformation is made, the codes can be compressed using a statistical algorithm. Bentley et al. have shown that MTF has similar performance to Huffman coding [4].

A number of variations of MTF have been proposed. MTF-1 moves symbols to the second position, unless they are already in that position. In that case, the symbol is moved to the first position. Move-ahead-k moves the symbol forward k positions. Wait-c-and-move will move a symbol to the front after it has seen it c times.

Burrows-Wheeler Transform

The Burrows-Wheeler Transform (BWT) [5] is a transformation algorithm that operates on strings of data. For a compression program that uses this algorithm, the transformation will usually be followed by MTF, RLE and a statistical method such as Huffman Coding or Arithmetic Coding.

If the string length is N, then a matrix M_1 of size $N \times N$ is formed. Note that this matrix is only conceptual and the actual implementation can be made in $O(N)$ space. Each row k is initially set to the string created by cyclically shifting the original block to the left by k steps. All rows are then sorted. The output of the BWT is the last column of the matrix and a row index I showing the row which contains the original block.

For example, consider the input block $S = [t, e, s, t, i, n, g]$. $|S| = 7$, so we create a 7×7 matrix using the method described in the above paragraph.

$$M_1 = \begin{pmatrix} t & e & s & t & i & n & g \\ g & t & e & s & t & i & n \\ n & g & t & e & s & t & i \\ i & n & g & t & e & s & t \\ t & i & n & g & t & e & s \\ s & t & i & n & g & t & e \\ e & s & t & i & n & g & t \\ t & e & s & t & i & n & g \end{pmatrix}$$

After sorting the rows in lexicographical order, we obtain the following matrix.

$$M_2 = \begin{pmatrix} e & s & t & i & n & g & \mathbf{t} \\ g & t & e & s & t & i & \mathbf{n} \\ i & n & g & t & e & s & \mathbf{t} \\ n & g & t & e & s & t & \mathbf{i} \\ s & t & i & n & g & t & \mathbf{e} \\ t & e & s & t & i & n & \mathbf{g} \\ t & i & n & g & t & e & \mathbf{s} \end{pmatrix}$$

From this we find $I = 5$ and the last column is the transformed block $bwt(S) = [t, n, t, i, e, g, s]$.

Reversing the transform is slightly more complicated. The input string L is sorted into a new string F. This string corresponds to the first row of the matrix M_2. We

construct a translation table T that shows the relation between the elements of L and F. This is shown in algorithm 3.

Input: L: The transformed string, T: The translation table, I: The index
Output: S: The original string
begin
    $t \leftarrow I$;
    for $i \leftarrow 0$ to $|L|$ do
        $S[|L| - i - 1] \leftarrow L[t]$;
        $t \leftarrow T[t]$;
    end
    return S;
end

Algorithm 3: *Reversion of Burrows-Wheeler Transform.*

A string transformed with BWT has the property of being locally homogeneous. This means that for any substring w of $bwt(S)$, the number of distinct symbols in w is likely to be small. This property is required for MTF to be effective.

Chapter 3

Design and Implementation

In this chapter I describe the design and implementation of a collection of decompression applications. I have implemented this software in Contiki in order to compare the characteristics of existing data compression algorithms when they are used on dynamically linkable modules to reprogram wireless sensor networks. The focus of this chapter is on the aspects of how the memory constraints of sensor nodes affect the design choices that have been made in this report.

Existing data compression applications are typically designed to achieve a high compression ratio on many different types of data in a moderate amount of time. This objective can be accomplished by using large memory areas to store higher order statistical models, transformation buffers, or search structures. Furthermore, they are designed to execute on personal computers, servers, and other types of machines that have resources that far exceed the hardware constraints imposed on sensor nodes.

In the context of wireless sensor networks, compression ratios are not necessarily directly related to the energy-efficiency of a compression algorithm. An extensive decompression procedure could offset the savings in radio transmissions. The memory access patterns of an algorithm also affect the energy consumption. For example, flash memory operations use considerably more energy than ordinary RAM accesses in the sensor nodes that are covered in this report. The flash memory is often divided into segments, where it is as costly to write one byte as it is to write a segment of bytes. In such a case, it is beneficial to buffer data in memory until a full segment can be written.

Although energy-efficiency is the primary goal, there are other issues that have an impact on the choice of the method. When compression algorithms are adapted to sensor nodes, the most important constraint is that of the low amount of memory. Algorithms that depend on sliding-window techniques or large transformation buffers can not hold all their state in memory, which implies that much of the state must be located in external flash memory. Special care must also be taken when selecting the internal data representation for the statistical models and the use of memory buffers. In order to reduce the energy costs and function call overhead of reading and writing to flash memory, as much data as possible should be buffered in memory.

Another important aspect is that the memory has to be shared between the Contiki kernel and a number of executing applications. Consequently, the decompression should if possible use significantly less memory than what is available. The design goals for the decompression software implementations in this report are therefore stated as follows:

Reduced energy consumption. The primary goal of using data compression for reprogramming sensor nodes is to reduce the energy consumption. Radio operations are significantly more expensive in terms of energy on the sensor nodes (section 4.3), which implies that the reduced radio traffic achieved by compression must successfully be traded off for the increased computational energy required by the decompression process.

Low computational overhead. The decompression process must not be demanding in computational needs as not to disrupt the ordinary tasks of the sensor node. It should use any cooperative measurements available in order to not monopolize the use of the micro-controller for too long time.

Small memory footprint. The software must operate with less than 10 kilobytes of memory and 64 bytes of flash memory, as is available in the Tmote Sky board. Preferably, there should be a large margin of the memory available for the operating system and the applications. Section 4.1 contains a detailed description of the targeted hardware platforms.

Small storage requirements. The software must be small in size because there is a very limited amount of program memory available in the sensor nodes.

The software is implemented in the C programming language because it gives fine-grained control over memory use and all the Contiki API:s have native C interfaces.

The remaining part of this chapter describes the design and implementation of a number of decompression algorithms, and how the memory-constrained environment in sensor nodes affect the design choices.

3.1 Integrating the Decompression Software in Contiki

The reprogramming process in Contiki is initiated by an operator who compiles the software at a base station. The software is then sent to a node that is connected by wire to the base station. This node will then send the software over-the-air to nodes close by with the help of a code propagation application, and the software will continue to propagate until it has been distributed to all nodes in the network.

When a sensor node receives fragments of a file, it is stored in flash memory by using the Contiki File System (CFS) interface. The CFS back-end writes the files to flash memory without intermediate buffering in memory. This implies that it is beneficial for the software to buffer I/O data in order to reduce the overhead of calling the read and write functions. Once the complete file has been received, the software is linked and loaded before executing it.

Each compression program implemented in this report contains a few bytes in the header of the file that identify the file format. Therefore, the receiving sensor node can choose to decompress the received software with the appropriate algorithm, or disregard the compression stage if the file is uncompressed. If it is compressed, the code propagation process sends a synchronous event to the appropriate decompression process. The associated event data consists of the first address of the file in flash memory, the file size and an address in flash memory to write to.

3.2 SBZIP: A BWT-based Compressor for Sensor Nodes

As part of this report, I have designed and implemented the SBZIP (Sensor Block Zip) compression software specifically to operate on memory-constrained systems such as sensor nodes. It is based on the Burrows-Wheeler transform, but differs from the original technique because the block sizes are very small, and the Huffman codes are constant. SBZIP divides the input file into blocks with a maximum size of 256 bytes. There is a trade-off between memory use and compression ratios, because more similar symbols can be grouped together with a larger block.

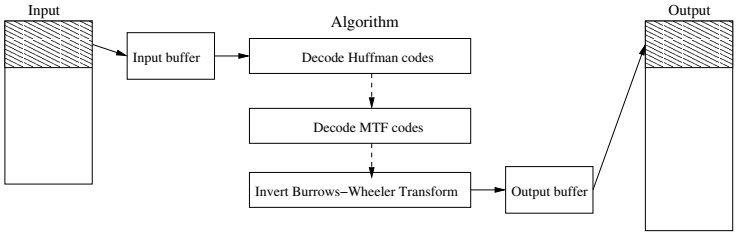


Figure 3.1: The stages and the use of I/O buffers on a single block.

Each block is operated on separately with three algorithms. The first step in the compression process is to do a Burrows-Wheeler transformation (section 2.2.4) on the block. Move-To-Front coding (section 2.2.4) is thereafter applied on the block, before doing the final Huffman coding (section 2.2.2). When decompressing, the respective decompression steps are taken in the reverse order.

The Huffman codes are generated with a fixed statistical model. The statistical model depends on the premise that for two symbols with integer representations S_1 and S_2, if $S_1 < S_2$, then $Pr(S_1) > Pr(S_2)$. Canonical Huffman codes are constructed using a binary minimum-heap. This technique is very useful for sensor nodes, because it uses $O(2n)$ space, where n is the number of symbols in the alphabet. Once the codes have been generated, they can be stored in static memory, because they can be reused in subsequent calls to the compressor.

The bottleneck of the compression is to sort the Burrows-Wheeler transformation matrices for all blocks, which is being done with insertion sort. The transform is an asymmetric algorithm because the decompressor only needs to sort the characters of a single string, instead of all the permutations of the string. This is accomplished with heapsort, which uses $O(N \log N)$ time on average and $O(1)$ space.

As shown in figure 3.1, SBZIP uses two memory buffers, each having the same size as the blocks. For each block, the input buffer is first filled with the block data from flash memory. The output buffer is where the final algorithm writes it output. This buffer is then copied to the next location in flash memory.

3.3 GZIP decompressor

One very popular general purpose compression program is GNU zip (GZIP), mainly because of its availability on many platforms and efficient compression. GZIP exclusively uses the DEFLATE algorithm (section 2.2.3) for compression. The first concern when implementing DEFLATE on memory-constrained systems is that sliding windows can be up to 32 kilobytes large, which is more than the available memory in the hardware platforms used for the evaluation (section 4.1). For this reason, we

store the larger part of the window in external flash memory. There is a trade-off between the amount of memory used in the buffers and the increased energy consumption, because a greater memory use will consequently lead to a smaller probability of having to read from flash memory. This is an issue that can significantly reduce the energy consumption of the decompressor. An alternative method is to use a cyclic buffer cache can be used to reduce the number of flash memory reads when copying referenced data.

The data structure and search method for the Huffman codes greatly affects the execution time of the decompressor. As a compromise between space and speed, the Huffman decoder uses lookup tables for codes that are at most seven bits long. Longer codes require a sequential search, but these are less likely to occur because the symbols with the highest probability have been assigned shorter codes.

I/O calls are made through function pointers in order to disassociate the algorithm from the system dependencies. This makes the software more easily portable between different operating systems and I/O services. The up-call interface consists of three functions: *gzCopy*, *gzInput* and *gzOutput*, and these functions use the CFS interface.

Memory Allocation

The decompressor uses a stack-based memory allocation scheme. This is beneficial on a memory-constrained system because extensive use of dynamic memory allocations on the heap will eventually make the heap fragmented. Avoiding fragmentation is important because all programs share the same address space in Contiki. The only exception to this is the Huffman trees, because they are used on several levels in the function call tree. It would therefore require an impractical design to allocate them on the stack. Static storage of Huffman trees is disregarded because the trees vary in size, and it would use memory even when the decompressor is not in use.

3.4 Arithmetic Coder

The arithmetic coder uses an adaptive order-0 model with symbol frequencies stored in 32-bit integers. Initially, 16-bit calculations were regarded because this is the native integer size of the MSP430 micro-controller that is included in both the boards used for the experiments. However, this led to a lower compression ratio while not increasing the speed of the computations enough to offset the lower compression ratio. Therefore, 32-bit integers were chosen for all the calculations in the statistical model and the intervals.

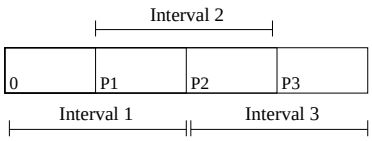


Figure 3.2: The scaling intervals for the arithmetic coder.

The integers that store the interval ends must be scaled [14] because they quickly converge as more symbols are encoded. The scaling operation takes place once the low and $high$ values are both within one of three predefined subintervals (figure 3.2)

of the maximum interval. The three points delimiting these intervals are $P_1 = 2^{29}$, $P_2 = 2 \times 2^{29}$, and $P_3 = 3 \times 2^{29}$.

While decoding a symbol, the arithmetic coder calculates the cumulative counts of the frequencies of all symbols up to and including the symbol being decoded. This is the most time-consuming task in the decoder. An alternative approach is to keep the counts in memory, but this would still require that the counts are incremented when updating the model for each processed symbol.

3.5 VCDIFF Decoder

We implement a decoder for the VCDIFF format (section 2.2.3) in order to compare how a method used to create compressed differences between two files performs when there is no source file available. The VCDIFF implementation uses external flash memory proportional to $O(M + N)$, where M is the size of the source file, and N is the size of the target file. The decoder also has to maintain an address cache to have more concise address representations. This cache uses a maximum memory of 1554 bytes and is the significant part of the memory consumption.

Several implementations exist to create VCDIFF files. We use the Xdelta software to generate the differences. Xdelta has been evaluated and compared with other difference approaches [19]. The results suggest that it is one of the most efficient methods for generating deltas that are not dependent on any predetermined information of the input.

3.6 Adapting Existing Decompressors for Sensor Nodes

In addition to implementing a selection of algorithms, I have also adapted two existing compression software to run on Contiki on sensor nodes. The usual procedure of these implementations is that the decoding function is called with a pointer to the starting memory location of the input data, a pointer to the corresponding output location and the size of the input. This requires that the block is loaded from flash memory, and that the output is later written to flash memory.

LZO1X

The LZO1X algorithm is based on LZ77, but is optimized for speed [27]. The LZO library consists of several compression algorithms. In order to use LZO1X on complete files, I have implemented a decompressor for the output format generated by the lzopack application. In order to keep the memory use adjusted to the constraints of the sensor nodes, we use a block size of 1024 bytes.

LZARI

The LZARI software uses a Ziv-Lempel technique combined with arithmetic coding. LZARI uses a ring buffer of 4096 bytes that I have changed to 128 bytes in order to fit on the sensor nodes. Also, it uses arrays for symbol frequencies, cumulative symbol frequencies, and cumulative frequencies for positions. Even with this downward adjustment of the ring buffer, the implementation requires more memory than what is available on the ESB, but fits well into the memory of the Sky board (see section 4.5 for a closer look on the memory use of the implementations).

Chapter 4

Evaluation

In this chapter I evaluate the energy consumption, the execution times, the compression ratios, the memory footprints and the storage size of the implementations. The two most important metrics are the energy consumption and the memory footprints. Firstly, the total energy consumed by the reprogramming process must be reduced in order to extend the lifetime of the sensor network. Secondly, the decompression software should use considerably less memory than the limit of the hardware platform, because the operating system and other executing applications must have room to operate as well.

4.1 Experimental Setup

The experiments are performed with two types of sensor nodes, namely the Moteiv Tmote Sky board [26] and the Scatterweb Embedded Sensor Board (ESB) [2]. The boards are shown in figure 4.1. Both of these boards use micro-controllers based on the Texas Instruments MSP430 16-bit RISC architecture. These micro-controllers have been designed to consume a current on the order of 1 to 5 milliamperes, which makes them suitable for sensor nodes.

The Scatterweb Embedded Sensor Board

The ESB is equipped with a MSP430F149 micro-controller, 60 kilobytes flash memory, 64 kilobytes EEPROM, 2 kilobytes RAM, and a RFM TR1001 short-range radio transceiver. In Contiki, the driver programs the transceiver to communicate at a rate of 9600 bits per second. Although the radio can operate at higher data rates, the packet loss will also increase, so this conservative data rate was chosen in Contiki. It includes a vibration sensor, a temperature sensor, and two infrared sensors.

The Tmote Sky

The Tmote Sky board has a MSP430F1611 micro-controller, 10 kilobytes of RAM and 48 kilobytes of internal flash memory. It also has 1 megabyte of external flash memory in which Contiki stores received software before it is linked and loaded into the internal flash memory. For radio communications, the Tmote Sky uses a Chipcon 2420 RF transceiver. The optional sensors are for temperature, light, and humidity.

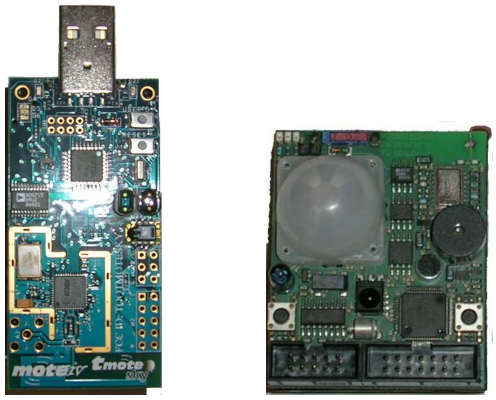


Figure 4.1: The sensor boards used for the experiments. The board shown on the left is the Tmote Sky, while the Scatterweb Embedded Sensor Board is shown on the right.

4.2 Data Set

For the experiments in this report, we evaluate the implemented decompressors and their corresponding compressors by applying them on a set of software chosen from the internal Contiki CVS repository at SICS. As shown in table 4.1, this set consists of applications, device drivers, and a larger operating system module. The software has been compiled for the MSP430 architecture by using version 3.2.3 of the GCC compiler. The files are stored in the ELF format.

Table 4.1: The software used to evaluate the compression algorithms. The file sizes are in bytes.

File	ELF size	CELF size	Description
ds2411	1548	712	Driver for the ds2411 sensor.
radio-test	2152	1253	Radio connectivity testing between nodes.
trickle	3560	2237	Implementation of the Trickle [24] protocol.
elfloader	4240	2996	ELF loader.
treeroute	8564	5938	A convergecast protocol.
uip	11704	7879	The uIP TCP/IP stack.

4.3 Energy Consumption

The power supply is connected to a circuit consisting of a sensor board and a 100 Ω resistor. The voltage drop is sampled with a digital oscilloscope. The energy consumption E_D is then given by $E_D = \frac{V_{in}}{R} \int_0^T v(t)\,dt$, where V_{in} is the input voltage, R is the resistance, T is the total time in which the samples were taken, and $v(t)$ is the voltage at time t. An example of the current consumption of the decompression process is shown in figure 4.2.

The energy consumption E of the complete reprogramming process has previously been modeled by Dunkels et al. [10] as

$$E = E_p + E_s + E_l + E_f, \tag{4.1}$$

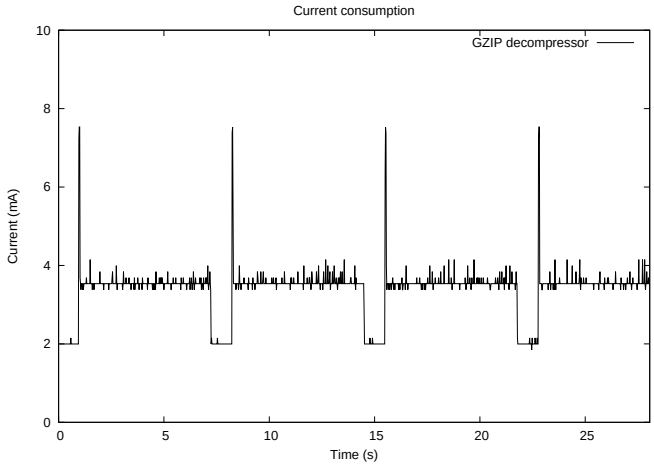


Figure 4.2: Current consumption of the GZIP decompressor when decompressing uip multiple times on the Tmote Sky. The spikes are caused by blinking a LED, and the low consumption after an iteration follows from having the process wait on a one second timer.

where E_p denotes the energy used for receiving and sending the file. E_s is the energy required to store the file in EEPROM, E_l is the energy consumed by the linking process. In this model, E_p is the most significant term because of the expensive radio operations. The cost of transferring the data is further magnified by the overhead of the chosen code propagation protocol. The transmission energy cost depends on the time that the sensor node has to keep the radio activated. For the continuing calculations, the terms E_l and E_f can be disregarded since they do not affect the performance of the compression stage, and are equal in both models.

A simplified model for estimating the energy consumption E_C with the effects of using data compression can be specified as

$$E_C = rE_p + (1 + r)E_s + E_D + E_l + E_f, \tag{4.2}$$

where E_D is the energy consumed for decompressing the file, and r represents the compression factor. Both of these terms depend entirely on the algorithm of choice and the file to be compressed. By subtracting E_C from E, we obtain the energy saved (or lost), denoted as S, from the use of data compression as

$$S = E - E_C = (1 - r)E_p - rE_s - E_D. \tag{4.3}$$

The energy cost of reception per byte of the CC2420 and the TR1001 radio transceivers have been measured to 0.0048mJ and 0.021mJ, respectively [10].

4.3.1 Energy Consumption of the Decompression

The energy required for decompressing the files in the data set is shown in table 4.2. Although the lower compression ratio of LZO1X implies that it has more bytes to process, it executes significantly faster than the other implementations because it has been optimized for speed. GZIP uses several times more memory on average for the decompression computation alone, but is clearly more energy-efficient than the other

implementations due to the relatively fast execution (section 4.6) and lower current consumption. The energy cost of the arithmetic decoder is negatively affected by symmetry of the arithmetic coding algorithm and the slow symbol lookups in the implementation.

Table 4.2: The energy E_D (in mJ) required for decompressing all files in the data set with different algorithms.

File	AC	GZIP	LZARI	LZO1X	SBZIP	VCDIFF
ds2411	72.9	11.9	31.6	2.1	38.0	34.5
radio-test	113.4	22.9	58.1	3.7	58.2	57.8
trickle	188.9	35.4	91.3	5.8	98.0	93.4
elfloader	247.2	49.6	112.8	6.9	134.2	111.0
treeroute	501.1	96.3	220.9	13.8	266.4	222.9
uip	645.6	124.0	295.8	18.6	341.1	308.3
Average	294.9	56.7	135.1	8.5	156.0	138.0

4.3.2 Radio Transmission Overhead

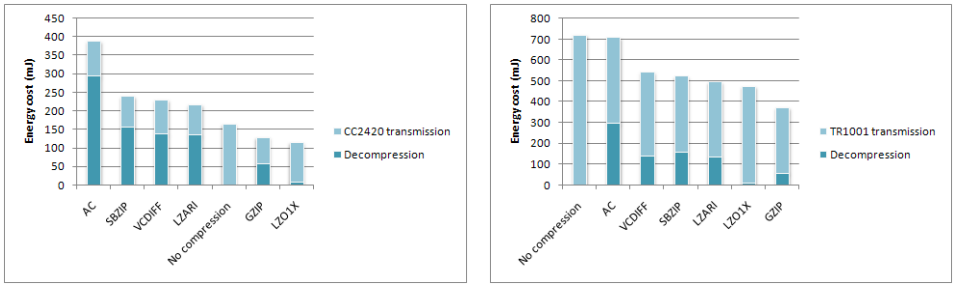
The choice of a code propagation protocol has a significant effect on the energy savings calculations because the protocol can induce a large transmission overhead. A high overhead implies greater energy savings when using data compression. This is because the E_p term increases while the other terms stay constant. In these calculations, I assume that the Deluge citedeluge04deluge protocol is being used because it is often referenced in research literature. Empirical data gathered from simulations suggest that Deluge causes nodes to receive approximately 3.35 times as many data packets as required [16]. Deluge uses 36 byte data packets that include a header of 13 bytes, thus it has a 50 % overhead for data packets.

Sensor networks with high packet loss affect the model similarly by causing an increase in the energy required to propagate the software due to retransmissions. Scheduled MAC protocols can be used to keep colliding transmissions to a minimum.

4.3.3 Energy Trade-offs

Lane and Campbell have analyzed the energy consumptions of the individual instructions in the MSP430, and the results show that the most common instructions use energy on the order of a few nanojoules [22]. These results, compared with the energy consumption of the radio transceiver, suggest that many thousands of instructions can be traded off for saving one byte of radio transmission. The decompression processes in this report, however, typically requires between one and ten seconds (section 4.6) of micro-controller activity.

The energy trade-offs between radio transmission and decompression is shown in figure 4.3(a) and 4.3(b). The transmission energy has been calculated by multiplying the estimated protocol overhead per byte (section 4.3.2) with the cost to receive one byte on the radio. LZO1X is significantly faster for decompression than the other implementations, and this is also the reason for why it consumes the least energy at the decompression stage. The costs for transmission are nevertheless higher than for the other implementations.



(a) Energy costs with the CC2420. *(b) Energy costs with the TR1001.*

Figure 4.3: The trade-off between energy for the radio traffic and the decompression computations.

Perhaps the most interesting result is that LZO1X is the most energy-efficient when the CC2420 is used for communication, despite having the lowest compression ratio. When the less energy-efficient TR1001 radio is used, however, the high compression ratio of GZIP makes it more energy-efficient than LZO1X. Compared to the base case of using no compression, the energy consumption is on average reduced by 47 % when using GZIP to distribute the software with the TR1001 radio, and 21 % when distributing it with the CC2420 radio.

4.4 Compression Ratios

The ELF files of the Contiki applications have certain characteristics that make them compressible to between 50 % and 60 % percent of the original size. Each ELF section contains statistical redundancies. For example, the instruction codes are located in the text section, and a small set of instructions are likely to occur much more often than others. The string table and symbol table typically contain English words, in which certain letters and sequences are much more common than others [30].

Table 4.3: The information entropy of the test files with three significant digits.

File	Entropy
elfloader	4.71
treeroute	4.68
uip	4.67
trickle	4.13
radio-test	3.81
ds2411	3.62
Average	3.89

The results obtained from measuring the information entropy (section 2.2) of the test files are shown in table 4.3. The average of 3.89 indicates that a compression of approximately 50 % can be achieved by an efficient compression software. This is confirmed by the results shown in table 4.4 and figure 4.4.

GZIP gives the best compression ratios of the measured implementations. SBZIP performs slightly better than average despite the small block size in use. LZO1X,

Table 4.4: Compressed file sizes (in bytes) of the different algorithms. The VCDIFF files have been generated with the xdelta3 software.

File	Original size	AC	GZIP	LZARI	LZO1X	SBZIP	VCDIFF
ds2411	1308	740	645	641	844	664	777
radio-test	2176	1146	1037	1066	1447	1020	1316
trickle	3560	1961	1542	1694	2200	1704	1977
elfloader	4240	2617	2014	2255	2860	2360	2551
treeroute	8564	5147	3789	4494	5720	4583	4853
uip	11704	6492	4754	5623	7345	5714	6236
Average	5259	3017	2297	2629	3403	2674	2952
Savings		42.6 %	56.3 %	50.0 %	35.2 %	49.1 %	43.9 %

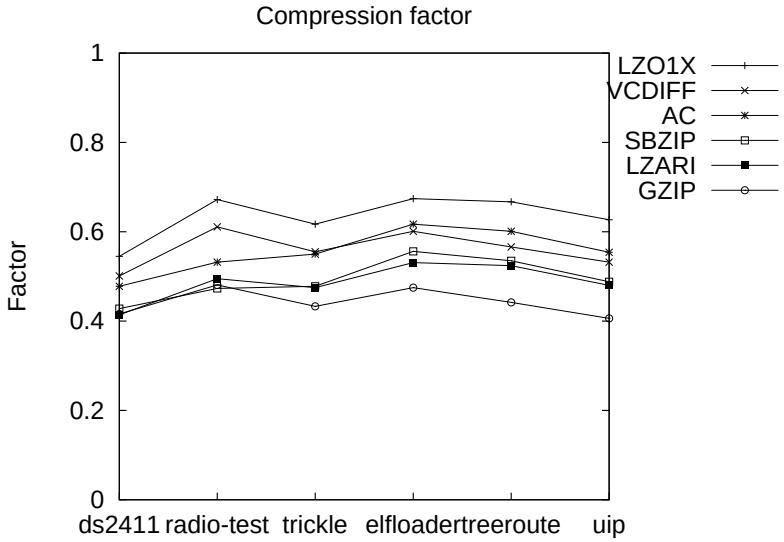


Figure 4.4: Comparison of the compression factors. The files are ordered by size with the smallest on the left side.

being optimized for speed rather than high compression ratios, does not surprisingly perform worse than the other compressors in this measurement.

4.5 Memory and Storage Requirements

Due to the memory-constrained environments in many types of sensor nodes, both the maximum RAM use and the internal flash memory footprint are relevant metrics when determining the feasibility of an implementation. Because the memory is shared with a Contiki kernel and a group of executing processes, the memory footprints should be kept significantly lower than the available memory. Additionally, since there is no memory protection on the nodes, a margin of safety is preferable.

The code size and memory footprints of each of the implementations is shown in table 4.5. The code size represents the ELF $.text$ section size that is stored in internal flash memory when the file has been dynamically linked. Memory footprints are measured by adding the static memory, the maximum stack memory, and the maximum heap memory used in the execution of a decompressor. The results show

Table 4.5: Memory footprints and code sizes (in bytes) for the programs.

Program	Memory	Code size
AC	552	1424
VCDIFF	1623	2890
SBZIP	1645	4358
LZO1X	2211	5146
GZIP	2357	7696
LZARI	2983	2328

that it is difficult to fit several of the implementations in the memory of the ESB, while also leaving enough space for other applications and the kernel. On the other hand, the Tmote Sky has considerably more memory than what is required to execute the decompression applications. The smallest implementation in both metrics is the arithmetic decoder. The decompression of AC is computationally intensive, but the memory requirements are low enough to make it feasible for the ESB.

4.6 Execution Times

The execution times as shown in figure 4.5 vary greatly depending on the implementation. The fastest is LZO1X because it does simple computations for each input byte. GZIP is surprisingly fast, given that it does both Huffman decoding and uses a sliding-window to copy from. The arithmetic decoder is clearly the slowest algorithm, because of the slow search required to find a symbol within a given interval. SBZIP is slower than the average because of the block sorting and the subsequent Huffman decoding.

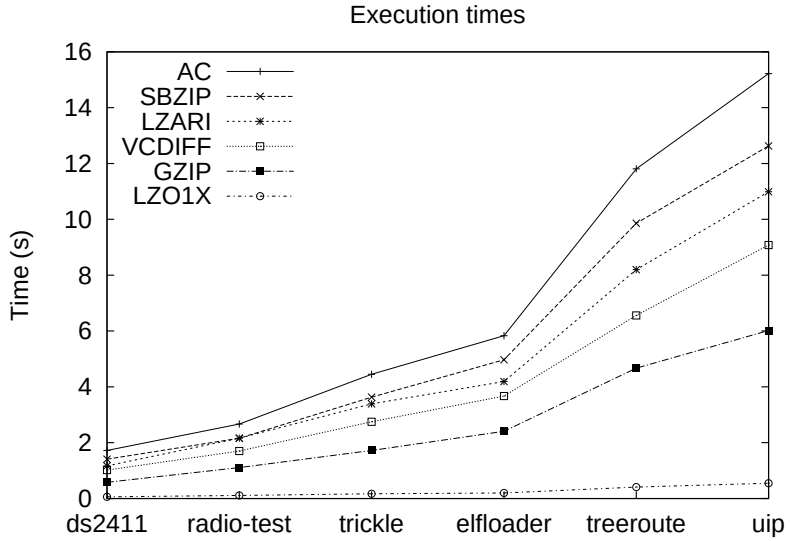


Figure 4.5: Execution times for decompression of the files in the data set. Measured on the MSP430F1611 processor.

Interrupts are processed transparently by the operating system during the execution of the decompression software. This increases the variance of the execution times. The standard deviations of the measured times are a maximum of 0.0082 for all of the implementations except LZARI. In the case of LZARI, the standard deviation was at most 0.3757, which was observed when measuring the time for decompressing the uip module. It is unclear why the measurements of LZARI have a higher variance. One plausible reason is that it uses unbuffered external flash memory more than the other implementations, which causes it to spend more time on waiting for external flash memory to be ready for I/O requests. Turning off interrupts would likely reduce the variance and also the execution times themselves, but this would also cause the measured times to be lower than what would be observed in a deployed system with interrupts on.

4.7 Combining CELF with Data Compression

CELF increases the entropy in the ELF files because a large number of bytes with value zero are removed when fields are scaled down from 32 to 16 bits. After converting the software in the test set to CELF, it was then compressed. The results in table 4.6 show that the the file sizes are reduced more if they are first converted to CELF prior to compressing them. The reduction is however on the order of 10 % for GZIP. Hence, the redundancies that CELF removes are almost as effectively removed by the compression algorithms.

Table 4.6: The average compression factors of using data compression on CELF files compared with using it on ELF files.

Program	Compressed CELF, in %	Compressed ELF, in %
AC	49.7	57.4
GZIP	39.5	43.7
LZARI	43.3	49.9
LZO1X	52.9	64.7
SBZIP	44.9	50.9
VCDIFF	48.9	56.1

LZO1X has the largest difference between the average compressed CELF size, and the average compressed size. Thus, the high energy-efficiency of LZO1X, as shown in section 4.3.3, can be substantially improved by converting the software to CELF before compressing it.

Chapter 5

Related Work

The energy-efficiency of using run-time dynamic linking with ELF modules has been examined by Dunkels et al. [10]. In addition, the authors compare the technique with using the Contiki Virtual Machine. This report extends their work by also examining the energy-savings obtained by compressing the ELF modules.

Sadler and Martonosi have developed the Sensor LZW (S-LZW) algorithm for compressing sensor data [29]. As the name suggests, it is a modification of LZW (section 2.2.3) that has been tailored for sensor nodes. They measure the energy consumption and find that their method can save significant amounts of energy. They also discuss some of the implementation issues with the sensor nodes. Furthermore, they compare small variations of S-LZW that use different dictionary sizes and also a data transformation method. This report differs from their work because I am mainly concerned with decompression of software. The sensor data that they use for their evaluation is considerably more compressible than the software modules that I use for the experiments in this report.

Several lossless general-purpose compression programs have been evaluated in terms of energy efficiency by Barr and Asanović [3]. They also examine some of the trade-offs between compression ratio and energy efficiency. The experiments were made on a StrongARM SA-110 system, which has a processor running at a clock rate of 233 MHz and 32 megabytes of memory. Unlike their work, I explore algorithms that must be feasible for typical sensor nodes, which can have an 16-bit MSP430 processor and approximately 10 kilobytes of memory. In particular, this report evaluates the energy savings that can be obtained when compressing software modules.

An evaluation of various delta compression algorithms to create deltas between full system images has been done by Konstapel [19]. Additionally, a technique for changing the layout of the software modules to make the delta compression more effective is proposed. Konstapel includes an approximate energy consumption analysis.

The choice of a *code distribution protocol* is also a relevant subject. The protocol must efficiently and reliably distribute the software to all nodes in the sensor network. Various protocols have been developed for this purpose, including Deluge [16], Trickle [24] and MOAP [31].

Chapter 6

Conclusions and Future Work

Dynamically linkable modules can be distributed in wireless sensor networks at a lower energy cost if they are compressed, and thus the lifetime of the sensor network is extended. The best overall energy-efficiency is achieved with GZIP. Compared to using no compression of the data set, GZIP saves on average 21 % when used with the CC2420 radio, and 47 % when used with the TR1001 radio. The energy savings are nevertheless strongly dependent on the algorithm of choice, the execution speed of the implementation, and the energy cost of radio traffic in relation to the energy cost of micro-controller activity on the sensor nodes.

When the energy cost for radio communication is relatively high, as is the case with the TR1001, the GZIP algorithm saves more energy because it has a clearly higher compression ratio, while the execution time is the second lowest. On the more energy-efficient CC2420 radio, the savings were rather small. A few of the algorithms actually increased the energy cost of reprogramming with this radio. One implication of using a more energy-efficient radio is that the decompression time becomes a significant factor in determining the energy savings because the difference between the current consumption of the radio and the micro-controller is less considerable. Because of its low execution times, LZO1X is therefore more energy-efficient with the CC2420 than the other algorithms.

The memory requirements of the implementations are low enough to be used on the Tmote Sky, but the limited memory of the ESB pose some problems for several of the implementations. SBZIP and VCDIFF are feasible for the ESB and are similar in all metrics except code size. Due to the code size being approximately between 5 % and 20 % of the available internal flash memory of the hardware platforms used in this report, this factor has less significance than the memory footprint. On the Tmote Sky we notice that LZO1X and GZIP have comparable energy-efficiency, but the clearly higher compression ratio of GZIP makes it more beneficial to use because of the likely shorter propagation time.

In addition to being more energy-efficient in most cases, a benefit of using data compression is that the size of the data to be distributed to the nodes is reduced by approximately 50 % when using GZIP, and hence the network operations are less disturbed in terms of radio traffic congestion caused by the epidemic nature of code propagation protocols. We do not however quantify this potential benefit in this report and leave it for future work. One more experiment that could be interesting is to examine whether it is beneficial to use adaptive statistical modeling for each individual section in ELF files. Since the sections have very different characteristics, it is probably worthwhile to reset the statistical model when a section border is passed.

Another future line of investigation is to examine how difference methods and data compression can be combined in a protocol to choose the most efficient reprogramming method based on whether it is a smaller update or a new module that should be propagated.

Bibliography

[1] I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. A survey on sensor networks, 2002.

[2] CST Group at FU Berlin. Scatterweb Embedded Sensor Board. Web page. Visited Nov. 6, 2006.
URL: http://www.scatterweb.com/

[3] K. Barr and K. Asanovic. Energy aware lossless data compression. In *Proceedings of the First International Conference on Mobile Systems, Applications, and Services (MobiSys 2003)*, San Francisco, CA, USA, 2003.

[4] J. Bentley, D. Sleator, R. Tarjan, and V. Wei. A locally adaptive data compression scheme. *Communications of ACM, Programming Techniques and Data Structures*, 29(4), April 1986.

[5] M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Systems Research Center, May 1994.

[6] TIS Committee. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2, May 1995.

[7] A. Dunkels. Protothreads web site. Web page. Visited Nov. 27, 2006.
URL: http://www.sics.se/~adam/pt/

[8] A. Dunkels. The Contiki Operating System. Web page. Visited Oct. 24, 2006.
URL: http://www.sics.se/~adam/contiki/

[9] A. Dunkels. uIP – a TCP/IP stack for 8- and 16-bit microcontrollers. Web page. Visited Nov. 13, 2006.
URL: http://www.sics.se/~adam/uip/

[10] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt. Run-time dynamic linking for reprogramming wireless sensor networks. In *Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys 2006)*, Boulder, Colorado, USA, November 2006.

[11] A. Dunkels, B. Grönvall, and T. Voigt. Contiki – a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors*, Tampa, Florida, USA, November 2004.

[12] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys 2006)*, Boulder, Colorado, USA, November 2006.

[13] C. Han, R. K. Rengaswamy, R. Shea, E. Kohler, and M. Srivastava. Sos: A dynamic operating system for sensor networks. In *MobiSYS '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*. ACM Press, 2005.

[14] P. Howard and J. Vitter. Practical implementations of arithmetic coding. Technical Report 92-18, Department of Computer Science, Brown University, April 1992.

[15] D. Huffman. A method for the construction of minimum-redundancy codes. In *Proceedings of the Institute of Radio Engineers*, volume 40, pages 1098–1101, September 1952.

[16] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proc. SenSys'04*, Baltimore, Maryland, USA, November 2004.

[17] J. Jeong and D. Culler. Incremental network programming for wireless sensors. In *Proceedings of the First IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks IEEE SECON*, October 2004.

[18] J. Jeong, S. Kim, and A. Broad. Network reprogramming. TinyOS documentation, 2003. Visited Apr. 6, 2006.
URL: http://www.tinyos.net/tinyos-1.x/doc/NetworkReprogramming.pdf

[19] M. Konstapel. Incremental Software Updates for Wireless Sensor Networks. Master's thesis, Delft University of Technology, May 2006.

[20] D. Korn, J. MacDonald, J. Mogul, and K. Vo. The VCDIFF Generic Differencing and Compression Data Format. RFC 3284, June 2002.

[21] J. Koshy and R. Pandey. Remote incremental linking for energy-efficient reprogramming of sensor networks. In *Proceedings of the second European Workshop on Wireless Sensor Networks (EWSN)*, 2005.

[22] N. Lane and A. Campbell. The influence of microprocessor instructions on the energy consumption of wireless sensor networks. In *Proceedings of Third IEEE Workshop on Embedded Networked Sensors (EmNets 2006)*, May 2006.

[23] P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. In *Proceedings of ASPLOS-X*, San Jose, CA, USA, October 2002.

[24] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proc. NSDI'04*, March 2004.

[25] P. Marrón, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, and K. Rothermel. Flexcup: A flexible and efficient code update mechanism for sensor networks. In *European Workshop on Wireless Sensor Networks*, 2006.

[26] Moteiv. Moteiv: wireless sensor networks. Web page. Visited Nov. 6, 2006.
URL: http://www.moteiv.com/

[27] M. Oberhumer. LZO real-time data compression library. Web page. Visited Dec. 19, 2006.
URL: http://www.oberhumer.com/opensource/lzo/

[28] N. Reijers and K. Langendoen. Efficient code distribution in wireless sensor networks. In *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, pages 60–67, 2003.

[29] C. Sadler and M. Martonosi. Data compression algorithms for energy-constrained devices in delay tolerant networks. In *Proceedings of SenSys 2006*, November 2006.

[30] C. E. Shannon. Prediction and Entropy of Printed English. *Bell Systems Technical Journal*, 30:50–64, 1951.

[31] T. Stathopoulos, J. Heidemann, and D. Estrin. A remote code update mechanism for wireless sensor networks. Technical Report CENS-TR-30, University of California, Los Angeles, Center for Embedded Networked Computing, November 2003.

[32] T. Voigt, H. Ritter, J. Schiller, A. Dunkels, and J. Alonso. Solar-aware Clustering in Wireless Sensor Networks. In *Proceedings of the Ninth IEEE Symposium on Computers and Communications*, June 2004.

[33] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT-23(3):337–343, May 1977.