Efficient Sensor Network Reprogramming through Compression of Executable Modules

Nicolas Tsiftes, Adam Dunkels, Thiemo Voigt
{nvt,adam,thiemo}@sics.se
Swedish Institute of Computer Science

***Abstract*—Software in deployed sensor networks needs to be updated to introduce new functionality or to fix bugs. Reducing dissemination time is important because the dissemination disturbs the regular operation of the network. We present a method for reducing the dissemination time and energy consumption based on compression of native code modules. Code compression reduces the size of the software update, but the decompression on the sensor nodes requires processing time and energy. We quantify these trade-offs for seven different compression algorithms. Our results show that GZIP has the most favorable trade-offs, saving on average 67% of the dissemination time and 69% of the energy in a multi-hop wireless sensor network.**

I. Introduction

Software in sensor networks needs to be updated for several reasons. The network may need new functionality that was not planned during the initial deployment. Bugs that have been discovered after the deployment may need to be fixed. Since it is not feasible to manually collect the deployed nodes, several methods for updating software using the built-in radio have been developed [1]–[5].

In this paper we present a method for sensor network software updates based on compression of executable native code modules. Native code modules allows updates at every level of the system, ranging from application programs to low-level hardware device drivers. By compressing the executable modules, we reduce the dissemination time and the energy consumption. We quantify the trade-offs in terms of dissemination time and energy, and decompression energy. We show that GZIP compression has the most favorable energy consumption, but that other compression mechanisms (e.g. SBZIP) have a smaller memory footprint.

Our contributions are threefold. Firstly, we show the feasibility of seven lossless data decompression algorithms, including the widely used GZIP, for embedded sensor nodes. Secondly, we quantify the trade-off between compression efficiency and energy consumption for seven compression algorithms and show that GZIP has the most favorable compression and energy properties. Thirdly, we evaluate code dissemination in terms of energy and completion time in a multi-hop sensor network testbed using Deluge. We quantify the impact on the completion time and the sensor node energy consumption.

We have implemented the decompression algorithms in the Contiki operating system running on the Tmote Sky [6] and the Scatterweb ESB [7] sensor nodes. Our implementations have a code footprint between 1.5 to 8 kilobytes and a RAM footprint of between 550 bytes and 3 kilobytes, which makes them feasible for memory-constrained sensor nodes. Our experimental evaluation shows that although most algorithms reduce the energy cost of reprogramming sensor networks with ELF modules, the savings of the algorithms vary considerably. The results show that in most scenarios, GZIP has the most favorable energy trade-offs since it combines a high compression ratio with fast execution times for decompressing the files in our data set. In addition, GZIP offers the largest reduction of the software size, on average by a factor of 0.44, which implies that GZIP-compressed software is disseminated faster than software compressed with the other algorithms.

The rest of this paper is structured as follows. We give a brief overview of relevant data compression algorithms in Section II. Section III describes the decompression algorithms that we implement. Our evaluation of the data decompression techniques is presented in Section IV. We describe related work in Section V, and conclude the paper in Section VI.

II. Background

Many different data compression algorithms exist. Their characteristics differ not only in compression ratios, but also regarding whether they are computationally intensive, or depend on large data structures. We review the four most common compression algorithms and describe the corresponding decompression algorithms.

A. Data Compression

Data compression algorithms are either lossless or lossy. Since we focus on compression of executable modules, the algorithms must be lossless. More specifically, they must guarantee that the original data is fully restored, because one erroneous bit could have a disastrous impact on the behavior of the software.

Statistical algorithms encode and decode data based on a statistical model of the data. The statistical modeling can be done in three ways. Constant models are pre-calculated and agreed on by the encoder and decoder. Semi-adaptive models are created by the encoder by making a pass over the file to calculate the model. In this case, the model must be sent alongside the encoded data in order for the decoder to be able to decode the data. Adaptive models are updated for each encountered symbol in the same way by the encoder and the decoder. The advantage of this is that there is no need to pack the model with the data, but the downside is that the modeler has to pass over many symbols before approaching optimality.

The encoder uses the statistical model to compress the symbols with an average code length close to the information entropy. The information entropy of order O of a random variable X is defined as $H(X) = -\sum_{i=1}^{n} p_i \log_2 p_i$, 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 the efficiency of the encoder.

Two different methods of statistical coding are Huffman coding and arithmetic coding. Huffman coding is a greedy algorithm that gives each symbol in the alphabet a value with a length that is an integral number of bits. The codes have unique bit prefixes so that they can be decoded unambiguously. By contrast, arithmetic coding uses a finite-length interval that is narrowed down for each encountered symbol in the input. Each symbol is assigned a sub-interval in proportion to its probability of occurrence in the input data. Once the input data has been processed, the lower end of the final interval becomes the compressed output.

Ziv and Lempel [8] developed two methods for compression commonly known as LZ77 and LZ78. LZ77 is a method for sliding window compression that takes advantage of information available in a history buffer by encoding a string that has been seen previously in the input. A number of variations exist for how this information is encoded. GZIP, for example, uses a pair that describes the backwards distance to an identical string and its length. LZ78 is an algorithm for data dictionary compression. This algorithm is similar to sliding window compression because it also uses references to previously encountered data. In this case, however, a data dictionary is built where the algorithm tries to match current input with an entry in the dictionary, and output the code of the entry.

The Burrows-Wheeler Transform [9] (BWT) performs a reversible transformation on a block of symbols in order to group together similar symbols. The transformation is not used by itself to compress the data, but is instead typically followed by Move-To-Front Coding [10] (MTF) and a statistical coder. The BWT on a string S of length N is conceptually a matrix in which row i contains the string S rotated i steps to the left. The rows are then sorted in lexicographic order, and a pair consisting of the last column L of the sorted matrix, and the row number I of the original string becomes the output.

B. Using Data Compression in Memory-Constrained Systems

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 and complex data structures to store higher order statistical models, transformations, or data dictionaries. Further, the compression algorithms are designed to operate on personal computers 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, efficient compression will significantly decrease code dissemination time, as we show in Section IV-C. However, the compression ratio is not necessarily directly related to the energy-efficiency of a compression algorithm. An extended decompression procedure could offset the savings in radio transmissions. The memory access patterns of an algorithm is an additional factor that affects the energy consumption. For example, operations on external flash memory consume considerably more energy than ordinary RAM accesses. Flash memory is typically divided into segments, where it is as costly to write one byte as it is to write a segment of bytes. Therefore, it is beneficial to buffer data in memory until a full segment can be written.

One additional issue that must be considered when adapting compression algorithms to sensor nodes is the extremely limited memory of the nodes. Algorithms that depend on sliding window techniques or large transformation buffers cannot hold all their state in memory, which implies that much of the state must be located in external flash memory. Special considerations must also be made when selecting the internal data representation for the statistical models and the use of I/O buffers. We do not consider compression software that uses higher-order models (e.g. PPMd) or large transformation matrices (e.g. bzip2), because the memory footprints of these algorithms are far beyond what is available in sensor nodes [11]. Another important aspect to consider is that the memory has to be shared between the operating system kernel and the active applications. Consequently the decompression should use significantly less memory than the available memory space of the sensor node.

III. Decompression Algorithms

Sensor network software is typically distributed from a resource-rich data sink where off-the-shelf compression implementations can be used. Hence, we focus on implementing the corresponding decompression algorithms for the very resource-constrained sensor nodes. In order to evaluate the efficiency of using data compression for code distribution in wireless sensor networks, we implement the decompression part of three widely used compression algorithms, namely arithmetic coding, DEFLATE (GZIP), and VCDIFF. We implement these algorithms from scratch because existing implementations of these algorithms are typically designed for computers that have significantly more memory than sensor nodes. To also study the performance and feasibility of using compression based on the Burrows-Wheeler Transform, we have designed and implemented SBZIP for BWT compression and decompression on sensor nodes.

Additionally, we have ported the existing compression algorithms LZARI, LZO1X, and S-LZW to the Contiki operating system. The first two algorithms are for general-purpose compression and work in many operating systems. Their low memory requirements make them viable to adapt directly to sensor nodes with minimal changes. S-LZW [12] is a compression algorithm specifically designed for sensor data.

A. SBZIP: BWT Compression for Sensor Nodes

The Burrows-Wheeler Transform performs optimally when a large set of symbols are grouped together in one trans-

formation. On sensor nodes, however, we are restricted to transformations on very small buffers due to the hardware limitations that exist to reduce monetary cost and energy consumptions. In order to evaluate BWT-based compression on sensor nodes, we have designed and implemented the SBZIP compression software (Sensor Block Zip) to operate on resource-constrained sensor nodes. Although SBZIP is based on the BWT algorithm, it differs from existing BWT-based compression software in that we use a 256 byte block size, which is considerably smaller than what available implementations use. For reasons explained in Section II-B, we select a block size that is equal to the external flash memory segment size. The small block size implies that fewer symbols can be grouped together by the BWT transformation, which decreases the achievable compression ratios. Additionally, we have chosen a set of constant Huffman codes to decrease processing time. This is yet another trade-off, but in this case we aim for a reduction in the processing time instead of a higher compression ratio.

Every block is compressed separately with a combination of three algorithms. The first step in the compression process is to do a Burrows-Wheeler transformation on the block. The transformed block is thereafter transformed again with Move-To-Front coding [10], before we do the final Huffman coding. When decompressing, the respective decompression algorithms are performed in the reverse order.

SBZIP generates the Huffman codes with a constant statistical model. This model depends on the premise that for two symbols with integer representations S_1 and S_2, if $S_1 < S_2$, then $Pr[S_1] > Pr[S_2]$. The tree for the canonical Huffman codes is constructed with a binary minimum-heap. This method is suitable for memory-constrained sensor nodes because it uses space of order $O(n)$, where n in this case is the number of symbols in the alphabet. Once the codes have been generated, they can be stored in static memory to be reused in subsequent calls to the compressor.

The performance bottleneck of the compression is to sort the transformation matrices for all blocks, which we do with insertion sort. BWT is an asymmetric algorithm because the decompression part only needs to sort the characters of a single string, instead of all the permutations of the string. We accomplish this with heapsort, which uses $O(n \log n)$ time on average and $O(1)$ space.

By reusing buffers for the stages of the algorithm, SBZIP only needs two main buffers of block size. For each block, the input buffer is first filled with the block data that is read from the external flash memory. The output buffer is where the final stage of the algorithm stores the result. Once the block has been compressed, it is copied from the output buffer to the next output location in the external flash memory.

B. GZIP Decompressor

In order to quantify how a widely used general-purpose compression algorithm performs on sensor nodes, we have implemented a decompressor for the GZIP standard format [13]. The implementation is made from scratch with an emphasis on low memory use. The GZIP specification only supports one compression algorithm–DEFLATE [14]. The DEFLATE algorithm divides the compressed data in variable-length blocks. Figure 1 shows a flowchart the DEFLATE operation on each block. The algorithm selects between three methods to use on each block individually based on the characteristics of the input data. Compressed blocks are created with a technique that combines sliding window compression [8] and Huffman Coding [15]. The canonical Huffman codes are prepended in the block in a format that is itself compressed. There are two different Huffman trees: one for literals and one for pairs of distance and length. Another less common method to compress blocks is based on fixed codes. This is inherently the same method as the one above, except that the Huffman codes are hard-wired in the software. We have implemented the fixed Huffman decoding. The third method is to store the block uncompressed. This is only useful when the statistical redundancies in the data are not large enough for the compressor to reduce the size of the output data.

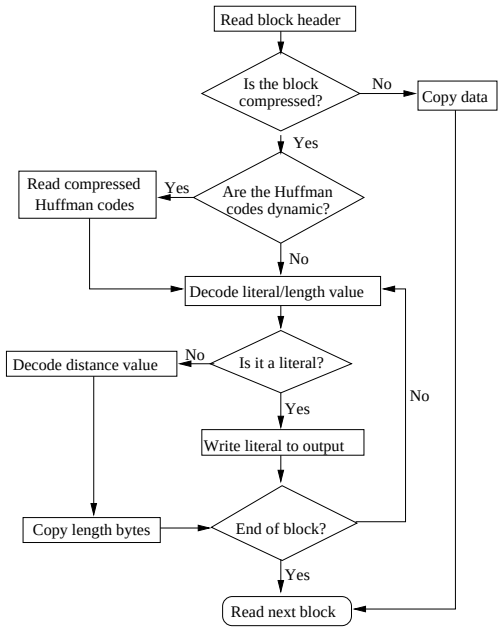

Fig. 1. A flowchart of the GZIP decompression algorithm.

DEFLATE uses a sliding window that can be up to 32 kilobytes large. Because this is much more than the available memory in most modern sensor nodes, the larger part of the window is stored 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 external flash memory. The implementation uses a 256 byte cache stored in RAM to provide fast copying of data that is close in the sliding window. This way we reduce the energy consumption of the decompressor.

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, our Huffman decoder uses lookup tables for codes that are at most 7 bits long. Longer codes require a time-consuming sequential search.

C. Arithmetic Decoder

We have implemented an arithmetic decoder with adaptive statistical modeling of order-0. Due to the choice of a simple data model, this implementation requires significantly less memory than the other implementations, but is more computationally intensive. The algorithm is also synchronous, which means that the decoding is approximately as complex as the encoding. Because the integers are of finite length, and the interval ends quickly converge, we use scaling [16] to output bits that are no longer needed for the calculations.

D. VCDIFF Decoder

In order to measure the performance of a file difference method when it is used for standalone compression, we have implemented a VCDIFF [17] decoder for sensor nodes. The VCDIFF format specifies how a description of the difference between two files is stored compactly. A VCDIFF encoder uses Ziv-Lempel sliding window compression and run-length encoding by conceptually concatenating the old and the new version of the file.

VCDIFF can be used for standalone compression by operating on an empty source file. The compression is not as efficient as that of GZIP, however, because it does not use a statistical compression method. Hence, it is possible to compress the deltas further by using a secondary compressor. For our experiments, we use the open source tool Xdelta to generate the VCDIFF files. Xdelta has been shown to be one of the most efficient VCDIFF encoders [18].

We have implemented a VCDIFF decoder for sensor nodes that uses external flash memory of order $O(m+n)$, where m is the size of the source file, and n is the size of the target file. The main part of the decoder's memory footprint is attributed to the address cache of 1554 bytes. The cache is needed to store addresses in a compact manner.

IV. Evaluation

To evaluate data compression for reprogramming sensor networks, we experimentally quantify the dissemination time and energy, the memory footprints, the compression factors, and the execution times of the different algorithms using a testbed of Tmote Sky nodes. We look at two scenarios with the first being a single-hop transfer setup. The purpose of this is to establish a base line of how efficient compression algorithms are when the protocol overhead is minimal. The second scenario is a 16-node multi-hop testbed running the Deluge protocol for distribution of software.

A. Experimental Method and Setup

The experiments are conducted with the six software modules shown in Table I. The selected set includes both applications and operating system modules, i.e., software that actually might be updated in a deployed sensor network.

TABLE I
THE SOFTWARE USED TO EVALUATE THE COMPRESSION ALGORITHMS. THE SIZES ARE IN BYTES.

Name	ELF file size	Entropy	Description
File A	1548	3.62	Driver for the ds2411 sensor.
File B	2152	3.81	Radio connectivity testing.
File C	3560	4.13	Trickle [2].
File D	4240	4.71	ELF loader.
File E	8564	4.68	A convergecast protocol.
File F	11704	4.67	The uIP TCP/IP stack.

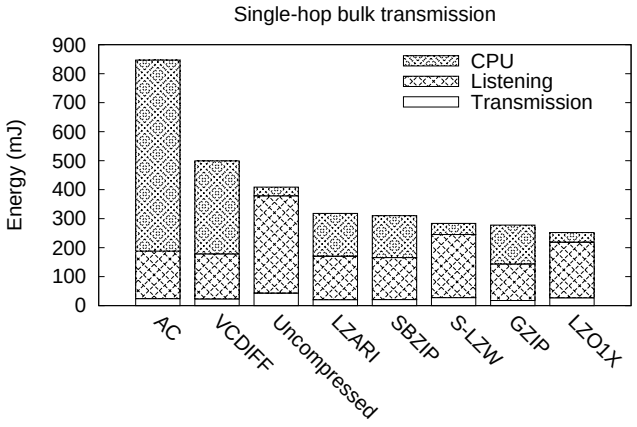

Fig. 2. The energy profile of a sensor node when receiving the file F in a single-hop bulk transmission using different compression algorithms. Listening represents the energy consumption of the radio when it is in receive mode.

To measure the energy consumption of the decompression process, we connect a power supply to a circuit consisting of a sensor node and a $100\,\Omega$ 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 the resistance, T the total time in which the samples were taken, and $v(t)$ the voltage at time t. We conduct the experiments on Tmote Sky nodes that are equipped with an MSP430F1611 micro-controller, 10 kilobytes of RAM, 48 kilobytes of internal flash, 1 megabyte of external flash, and a Chipcon CC2420 radio transceiver.

B. Single-hop Transmission

To measure the effect of using compression with a low protocol overhead, we set up an experiment consisting of two Tmote Sky nodes that communicate in a single hop. One node is the code originator and sends the complete file to the other node. We use the reliable unicast bulk transfer protocol provided by the Rime communication stack [19]. We measure the energy consumption of the receiving node using Contiki's energy estimation mechanism [20]. Figure 2 shows that when the communication overhead is very low, the savings are reduced and some of the algorithms are not energy-efficient, even when compared to using no compression.

TABLE II
AVERAGE TIME (S) AND ENERGY (MJ) USED FOR DISSEMINATION WITH DELUGE IN A 16-NODE MULTI-HOP TESTBED.

Algorithm	File size	Time	Energy consumption
Uncompressed	5259	556	16609
LZO1X	3403	364	10914
GZIP	2297	185	5191

C. Data dissemination in a multi-hop network

In multi-hop sensor networks, the code distribution process is affected by a range of variables that are non-trivial to model or simulate accurately. This includes the interaction between different network layers, such as MAC protocols, transport protocols, and routing, but also the effects of the shared broadcast medium and collision rates. To obtain a holistic view over the code distribution and the impact of data compression, we use a sensor network testbed of 16 Tmote Sky nodes using the Contiki operating system. The maximum number of hops is four. The radio medium access is controlled according to the X-MAC [21] protocol with a 10% duty cycle, as is currently the default in Contiki. All nodes run the Deluge protocol [4] for data dissemination. Deluge uses a set of constants to steer the behavior of the protocol. We set these constants as follows: $S_{pkt} = 32$, $S_{page} = 128$, $\omega = 8$, $\lambda = 2$, $\tau_l = 2$, and $\tau_h = 32$.

For the measurement we compare the two most efficient algorithms in a single-hop transfer (Section IV-B with the base case of using uncompressed code.

1) Dissemination Time: Short data dissemination times are either critical or desirable depending on the type of sensor network deployment. A prolonged dissemination phase disturbs the network operation considerably because of the flooding nature of data dissemination protocols. An example of this is in the emerging area of building automation [22] where expensive equipment is controlled using wireless sensor networks. Experiments by Hui and Culler [4] show a close to linear growth of data dissemination time in relation to the data size when using Deluge. Data dissemination protocols cannot have data rates close to the capacity of the medium because of broadcast transmission on a single channel, and suppression delays. Deluge restricts a sending node to only send one page at a time. Thus, nodes that request another page will have to wait for the transfer of the complete page to finish. As expected, the results in Table II show that the time reduction is larger than the compression factor. The efficient compression of GZIP gives 67% shorter dissemination time in our multi-hop testbed. The dissemination of the LZO1X files requires significantly more time, and saves only 34.6%.

2) Dissemination Energy: We use Contiki's software-based energy estimation mechanism [20] on every node to determine the lowest, the average and the highest energy consumption for the nodes. The transfer energy measurement is started when a node receives a Deluge profile message for a new version of a file. Once all new Deluge pages have been received, the measurement is stopped. The on-line energy estimator is then used to calculate the cumulative time of the activities of the components on the sensor node. These are multiplied by electrical current of each component, respectively, and our voltage supply to the Tmote Sky, which is 3 V. The resulting times shown in Table II correlate much with the time. This is because the idle listening of the radio is the key contributor to the energy consumption. Due to the compression efficiency of GZIP, the small amount of bytes that must be disseminated also results in a reduction of approximately $2/3$ of the energy consumed when using no compression.

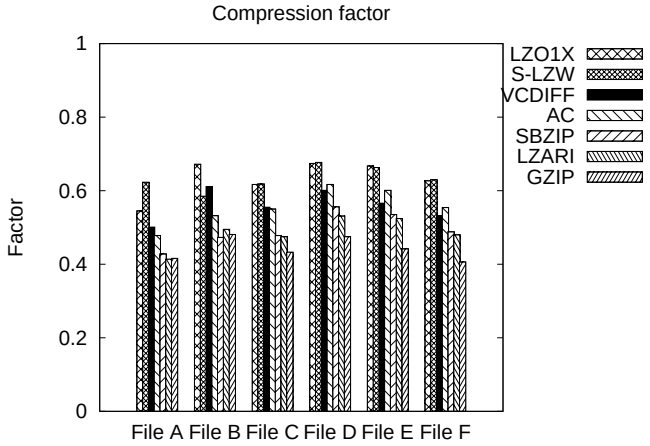

Fig. 3. Comparison of the compression ratios. The files are ordered by size with the smallest on the left side.

D. Compression Factors

Dynamically linkable modules in the ELF format [23] typically contain statistical redundancies in the most significant sections which contain instructions, data, relocation, and symbols. For example, in many instruction formats, a small set of instructions are likely to occur much more often than others (e.g., a stack operation compared with an operation that writes to a control register). The string table and symbol table usually contain English words, in which certain letters and sequences are much more common than others. Therefore they can be compressed efficiently [24]. The results obtained from measuring the information entropy of the test files are shown in Table I. 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 in Table III.

Figure 3 shows that GZIP has the highest compression ratios of all algorithms for almost all files. SBZIP performs slightly better than most of the other algorithms despite using a very small block size for the BWT. As expected, LZO1X that is optimized for speed rather than high compression ratio has the lowest compression ratio of the algorithms in this experiment.

E. CELF Compression

CELF [1] is an executable software format that, unlike ELF, is designed for 16-bit computing architectures. The purpose of CELF is to remove redundant space in the 32-bit data fields

TABLE III
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	S-LZW	SBZIP	VCDIFF
File A	1308	740	645	641	844	816	664	777
File B	2176	1146	1037	1066	1447	1274	1020	1316
File C	3560	1961	1542	1694	2200	2203	1704	1977
File D	4240	2617	2014	2255	2860	2873	2360	2551
File E	8564	5147	3789	4494	5720	5681	4583	4853
File F	11704	6492	4754	5623	7345	7379	5714	6236
Average	5259	3017	2297	2629	3403	3371	2674	2952
Savings		42.6%	56.3%	50.0%	35.2%	36.0%	49.1%	43.9%

TABLE IV
THE AVERAGE FILE SIZES IN RELATION TO THE ORIGINAL FILES WHEN COMPRESSION IS USED ON CELF FILES, COMPARED WITH USING IT ON ELF FILES.

Program	Compressed CELF	Compressed ELF
AC	49.7%	57.4%
GZIP	39.5%	43.7%
LZARI	43.3%	49.9%
LZO1X	52.9%	64.7%
S-LZW	54.9%	67.0%
SBZIP	44.9%	50.9%
VCDIFF	48.9%	56.1%

of the ELF format. After compilation, the ELF files produced by a compiler can be converted to CELF.

A conversion to CELF increases the average entropy of the files in the test set to 5.65. The reason for this is that a large number of null bytes are removed. Uncompressed CELF reduces the files to 66.6% of their original size. We have examined whether compression in combination with CELF yields considerably better results than using compression alone. The results in Table IV show that the file sizes are reduced more if they are first converted to CELF prior to compressing them, but the reduction is on the order of 10% for GZIP. Hence, the specific redundancies that CELF removes are almost as effectively removed by the compression algorithms.

The greatest benefit of converting the files to CELF first is drawn by LZO1X. Thus, the high energy-efficiency of LZO1X, as shown in Section IV-B, can be improved substantially by converting the software to CELF before compressing it. The benefit of CELF, however, is limited to modules stored in the ELF format, whereas general-purpose data compression algorithms have the advantage of yielding slightly larger savings without being dependent on the software object format. GZIP also achieves the best compression on other types of software formats. For example, GZIP compresses a binary image of the TinyOS Blinker application stored in IHEX format to 27.8% of its original size.

F. Energy Model

The energy cost for the reprogramming process with dynamically linkable modules has previously been modeled as

$$E = E_p + E_s + E_l + E_f, \tag{1}$$

where E_p denotes the energy used for receiving and sending the file. E_s is the energy required to store the file in external flash memory, E_l is the energy consumed by the linking process, and E_f is the energy used for writing the linked software in internal flash memory [1]. 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. For the following 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

$$E_C = rE_p + (1 + r)E_s + E_D + E_l + E_f, \tag{2}$$

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

$$S = E - E_C = (1 - r)E_p - rE_s - E_d. \tag{3}$$

1) Decompression Energy: We have measured the energy consumption for decompressing the files in the data set with a digital oscilloscope. The results are shown in Table V. Although the lower compression ratio of LZO1X implies that it has more bytes to decompress, it executes significantly faster than the other algorithms because it has been optimized for speed. In most scenarios, GZIP consumes less energy than the other algorithms due to the relatively fast execution (Section IV-G) and a low current draw. Figure 4 shows how the current is affected when decompressing File F in the test set with the GZIP decompressor. The effects of reading or writing to external flash have a marginal impact on the current draw, compared with the activity of the micro-controller. The energy cost of the arithmetic decoder is negatively affected by symmetry of the arithmetic coding algorithm and the slow symbol searches in the implementation.

TABLE V
THE ENERGY E_D (IN MJ) REQUIRED FOR DECOMPRESSING THE FILES IN THE DATA SET WITH DIFFERENT ALGORITHMS.

File	AC	GZIP	LZARI	LZO1X	S-LZW	SBZIP	VCDIFF
File A	72.9	11.9	31.6	2.1	3.0	38.0	34.5
File B	113.4	22.9	58.1	3.7	4.2	58.2	57.8
File C	188.9	35.4	91.3	5.8	6.7	98.0	93.4
File D	247.2	49.6	112.8	6.9	8.7	134.2	111.0
File E	501.1	96.3	220.9	13.8	15.4	266.4	222.9
File F	645.6	124.0	295.8	18.6	22.1	341.1	308.3
Average	294.9	56.7	135.1	8.5	20.0	156.0	138.0

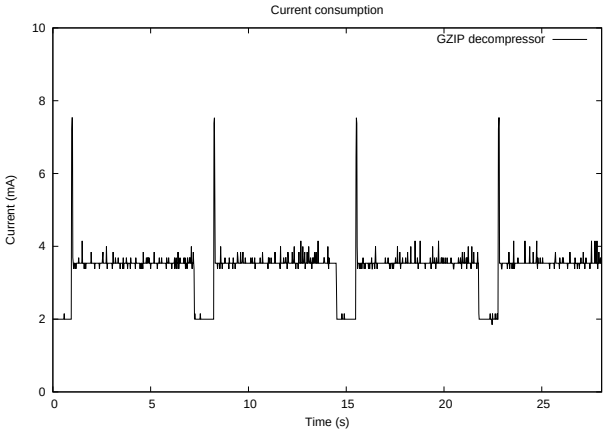

Fig. 4. Current consumption of the GZIP decompressor when decompressing File F repeatedly on the MSP430F1611 micro-controller. The spike before each iteration is caused by blinking the LEDs, and the low points are caused by having the process wait for a timer.

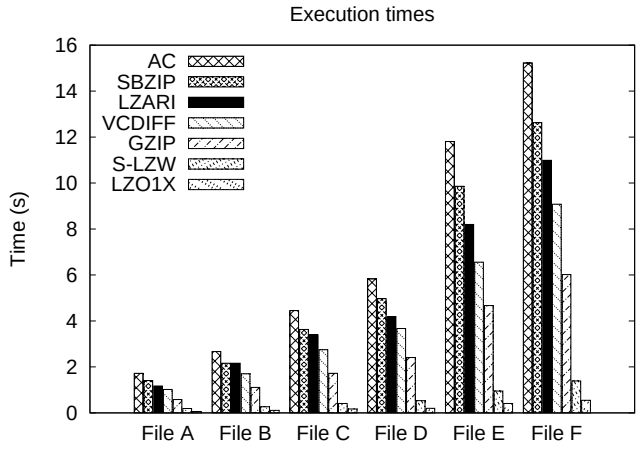

Fig. 5. Execution times for decompression of the files in the data set. Measured on the MSP430F1611 processor.

G. Execution Times

A long-running decompression process causes the CPU to be in active mode and consume energy that in some cases can be larger than the radio energy savings. Hence, to minimize the on-board energy consumption not caused by the radio, it is important that the decompression is fast.

We have measured the execution times by decompressing the files in the test set five times with each algorithm and then calculated the average of the execution times. Figure 5 demonstrates that the execution times vary greatly depending on the algorithm.

The fastest algorithm is the LZO1X because it only performs simple computations for each input byte. GZIP gives a high performance, even though it does both Huffman decoding and uses a sliding window to copy from. The high speed of the GZIP decompression can be attributed partly to the copying from the sliding-window, which reduces the number of symbols to decode. The more computationally intensive arithmetic decoder is clearly the slowest algorithm, because of the slow search that is required to find a symbol within a given interval. SBZIP is slower than most of the other algorithms mainly because of the block sorting.

Even though the execution times are several seconds on average for most of the algorithms, the decompression does not necessarily have to monopolize the micro-controller. The Contiki operating system provides the possibility to execute the applications in separate threads that can yield the control back to the scheduler at arbitrary points [25].

H. Memory Footprints

Sensor nodes are typically constrained with memory ranging from 1 to 10 kilobytes, and internal flash memory that is an order of magnitude larger. The memory space is shared with the operating system kernel and a set of processes. For this reason it is important that the decompression software is optimized to use significantly less memory than what is available on the nodes.

The code size and memory footprints of each of the implementations are shown in Table VI. The code size represents the ELF $.text$ section size that is stored in internal flash memory when the file has been dynamically linked. The memory footprint is the sum of the static memory, the maximum stack memory, and the maximum heap memory that is used while decompressing data. The results show that several of the algorithms have memory footprints that exceed the available memory in the more resource-constrained sensor nodes such as the ESB. On the other hand, the Tmote Sky has considerably more memory than what is required to execute all of the decompression applications. The smallest implementation in both metrics is the arithmetic decoder. The decompression of AC is computationally intensive, but the memory requirements

TABLE VI
MEMORY AND CODE SIZES (IN BYTES) FOR THE PROGRAMS.

Program	Memory footprint	Code size
AC	552	1424
VCDIFF	1623	2890
SBZIP	1645	4358
LZO1X	2211	5146
GZIP	2357	7696
S-LZW	2618	2564
LZARI	2983	2328

are low enough to make it feasible for the ESB. The S-LZW memory footprint is 2618 bytes when a dictionary of 512 entries is used, and every additional entry requires four bytes [12].

I. Summary of the Results

GZIP provides the highest compression ratio with the implication that it reduces the radio dissemination energy the most. However, our decompression implementation for GZIP takes longer time to decompress a file than both LZO1X and S-LZW. This makes the energy trade-off less favorable when the energy consumption on average for radio communication is low.

In a single-hop transfer, LZO1X is slightly more energy-efficient than GZIP. This is because of the optimized decompression algorithm in LZO1X, and the low communication overhead. S-LZW is comparable with LZO1X in terms of compression factor and energy consumption, but S-LZW saves less energy in all our measurements. However, when the software is disseminated in a multi-hop network with Deluge, GZIP is significantly more time and energy efficient. In this type of setting where the communication overhead is large, the compression factor of the algorithm is more important than the decompression time on each node.

SBZIP has a slightly higher energy consumption in the single-hop transmission experiment than S-LZW, but has the advantages of having a significantly smaller memory footprint and a higher compression ratio. LZARI has similar results with SBZIP in most of our measurements. However, it requires the largest memory footprint. The arithmetic coder's memory footprint is only a fraction of the others', which makes it feasible to use on more resource-constrained platforms, despite being less energy-efficient.

V. RELATED WORK

Due to the importance of being able to reprogram sensor networks, researchers have investigated into modular operating systems such as Contiki [25] and SOS [3] where one of the main arguments for modularity and dynamic code updates is that in a system statically linked at compile time "code updates become more expensive since a whole system image must be distributed" [3].

Furthermore, a number of energy-efficient code distribution mechanisms have been developed, including Deluge [4], Trickle [2] and MOAP [26]. An orthogonal line of investigation has been into methods for reprogramming sensor nodes using image replacements [27], virtual machines [28], version deltas [5], [29]–[31], and dynamically linkable modules [1].

The use of run-time dynamic linking of ELF modules has received attention [1] and both the Contiki [25] and Mantis [32] operating systems now provide dynamic linking based on the ELF format. ELF is also the default object file format produced by the GCC utilities and a number of standard software utilities for manipulating ELF files exist [1]. Dynamically linkable software stored in the ELF format is however quite large compared with pre-linked modules. Since the radio energy consumption of sensor nodes is typically an order of magnitude higher than that of the micro-controller [33], distributing relatively large files to all sensor nodes in a network is not energy-efficient. Dunkels et al. have for this reason provided a more compact alternative to ELF called CELF, Compact ELF [1]. Our work focuses on the compression of files in these two code formats.

Several lossless general-purpose compression programs have been evaluated in terms of energy efficiency by Barr and Asanović [11]. 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. In contrast with their work, we explore algorithms that must be feasible for typical sensor nodes, which may have a 16-bit MSP430 processor and approximately 10 kilobytes of memory. In particular, this paper evaluates the energy-savings that can be obtained when compressing software modules.

Lekatsas and Wolf [34] propose two algorithms for machine code compression in embedded systems, Semiadaptive Markov Compression (SAMC) and Semiadaptive Dictionary Compression (SADC). Their experimental evaluation shows that their algorithms have compression ratios similar to the Unix tool *compress*, but lower than GZIP. Our paper differs from their work because we compress all segments in the executable file, which include data, relocation information, symbol tables, and string tables in addition to machine code.

Sadler and Martonosi have developed the Sensor LZW (S-LZW) algorithm for compressing sensor data [12]. S-LZW is a modification of the LZW algorithm that has been tailored for sensor nodes. They measure the energy consumption and find that their method can save substantial 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 paper differs from their work because we are mainly concerned with decompression of software. The sensor data that they use for their evaluation is considerably more compressible than ELF modules.

VI. CONCLUSIONS

Compressing native code modules is a time and energy efficient approach to reprogram sensor networks. The time reduction when using the Deluge protocol in a multi-hop network is significant when an efficient compression algorithm such as

GZIP is used. However, the energy savings are also strongly dependent on the choice of the data compression algorithm, and the radio energy cost in relation to computation energy on the sensor nodes. Additionally, the ability of the dissemination protocol to transfer data in an energy-efficient manner affects the savings given by the compression algorithms. A high communication overhead makes the compression ratio of the algorithm more important.

The importance of evaluating several algorithms is highlighted by our discovery that a few of the implemented algorithms actually increase the energy consumption on a single-hop basis. When designing compression software for sensor networks, it becomes evident from the results in this paper that an algorithm with a high compression ratio is not necessarily energy efficient if the on-board processing time is considerable.

Our experiments show that the GZIP decompressor is the most energy-efficient algorithm when used in a multi-hop network with the Deluge protocol for data dissemination. The dissemination time and energy savings were approximately 67% in our testbed experiment. This implies that the service interruption during the reprogramming phase is reduced significantly, with the additional benefit of reducing the energy consumption compared to using no compression. Even though LZO1X compresses the files less than the majority of the other algorithms, it is the most energy-efficient algorithm in a single-hop transfer. The reason for this is the low energy cost per byte for transmission, and the low execution time of the decompression software. When communication devices with higher energy costs per transmitted byte are used, more energy is saved by using GZIP than with the other compression algorithms, even when the protocol overhead is low.

Acknowledgements

This work was financed by VINNOVA, the Swedish Agency for Innovation Systems, and the Uppsala VINN Excellence Center for Wireless Sensor Networks WISENET, also partly funded by VINNOVA.

References

[1] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt, "Run-time dynamic linking for reprogramming wireless sensor networks," in *ACM Conference on Networked Embedded Sensor Systems (SenSys 2006)*, Boulder, USA, Nov. 2006.

[2] 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*, Mar. 2004.

[3] C. Han, R. K. Rengaswamy, R. Shea, E. Kohler, and M. Srivastava, "SOS: A dynamic operating system for sensor networks," in *MobiSys '05*, 2005.

[4] 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, Nov. 2004.

[5] N. Reijers and K. Langendoen, "Efficient code distribution in wireless sensor networks," in *Proc. WSNA'05*, 2003.

[6] J. Polastre, R. Szewczyk, and D. Culler, "Telos: Enabling ultra-low power wireless research," in *Proc. IPSN/SPOTS'05*, Los Angeles, CA, USA, Apr. 2005.

[7] J. Schiller, H. Ritter, A. Liers, and T. Voigt, "Scatterweb - low power nodes and energy aware routing," in *Proceedings of Hawaii International Conference on System Sciences*, Hawaii, USA, 2005.

[8] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. IT-23, no. 3, pp. 337–343, May 1977.

[9] M. Burrows and D. Wheeler, "A block-sorting lossless data compression algorithm," Digital Systems Research Center, Tech. Rep. 124, May 1994.

[10] J. Bentley, D. Sleator, R. Tarjan, and V. Wei, "A locally adaptive data compression scheme," *Communications of ACM, Programming Techniques and Data Structures*, vol. 29, no. 4, Apr. 1986.

[11] K. Barr and K. Asanovic, "Energy aware lossless data compression," in *First International Conference on Mobile Systems, Applications, and Services (MobiSys 2003)*, San Francisco, CA, USA, 2003.

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

[13] P. Deutsch, "GZIP file format specification version 4.3," Aladdin Enterprises, RFC 1952, May 1996.

[14] ——, "DEFLATE Compressed Data Format Specification version 1.3," Aladdin Enterprises, RFC 1951, May 1996.

[15] D. Huffman, "A method for the construction of minimum-redundancy codes," in *Proceedings of the Institute of Radio Engineers*, vol. 40, Sep. 1952, pp. 1098–1101.

[16] P. Howard and J. Vitter, "Practical implementations of arithmetic coding," Brown University, Technical Report 92-18, Apr. 1992.

[17] D. Korn, J. MacDonald, J. Mogul, and K. Vo, "The VCDIFF Generic Differencing and Compression Data Format," Internet Engineering Task Force, RFC 3284, Jun. 2002.

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

[19] A. Dunkels, F. Österlind, and Z. He, "An adaptive communication architecture for wireless sensor networks," in *Proceedings of the Fifth ACM Conference on Networked Embedded Sensor Systems (SenSys 2007)*, Sydney, Australia, Nov. 2007.

[20] A. Dunkels, F. Österlind, N. Tsiftes, and Z. He, "Software-based online energy estimation for sensor nodes," in *Proceedings of the Fourth Workshop on Embedded Networked Sensors (Emnets IV)*, Cork, Ireland, Jun. 2007.

[21] M. Buettner, G. V. Yee, E. Anderson, and R. Han, "X-MAC: a short preamble MAC protocol for duty-cycled wireless sensor networks," in *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, Boulder, Colorado, USA, 2006, pp. 307–320.

[22] A. Bonivento, L. Carloni, and A. Sangiovanni-Vincentelli, "Platform based design of wireless sensor networks for industrial applications," in *Design Automation and Test in Europe*, Munich, Germany, 2006.

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

[24] C. E. Shannon, "Prediction and entropy of printed english," *Bell Systems Technical Journal*, vol. 30, pp. 50–64, 1951.

[25] A. Dunkels, B. Grönvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *Workshop on Embedded Networked Sensors*, Tampa, Florida, USA, Nov. 2004.

[26] T. Stathopoulos, J. Heidemann, and D. Estrin, "A remote code update mechanism for wireless sensor networks," UCLA, Center for Embedded Networked Computing, Tech. Rep. CENS-TR-30, Nov. 2003.

[27] J. Jeong, S. Kim, and A. Broad, "Network reprogramming," TinyOS documentation, 2003, visited 2006-04-06. [Online]. Available: http://www.tinyos.net/tinyos-1.x/doc/NetworkReprogramming.pdf

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

[29] J. Jeong and D. Culler, "Incremental network programming for wireless sensors," in *IEEE SECON*, Oct. 2004.

[30] J. Koshy and R. Pandey, "Remote incremental linking for energy-efficient reprogramming of sensor networks," in *Proc. EWSN'05*, 2005.

[31] P. Marrón et al., "Flexcup: A flexible and efficient code update mechanism for sensor networks." in *European Workshop on Wireless Sensor Networks*, 2006.

[32] E. Trumpler, V. Krunic, and R. Han, "NodeMD: Diagnosing node-level faults in remote wireless systems," in *MOBISYS '07*, San Juan, Puerto Rico, Jun. 2007.

[33] V. Raghunathan, C. Schurgers, S. Park, and M. Srivastava, "Energy aware wireless microsensor networks," *IEEE Signal Processing Magazine*, vol. 19, no. 2, pp. 40–50, Mar. 2002.

[34] H. Lekatsas and W. Wolf, "Code compression for embedded systems," in *Design Automation Conference*, 1998, pp. 516–521.