Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 10 (2012) 179 – 187

Procedia Computer Science

The 3rd International Conference on Ambient Systems, Networks and Technologies (ANT)

Energy-aware reprogramming of sensor networks using incremental update and compression

Milosh Stolikj[a], Pieter J.L. Cuijpers[a], Johan J. Lukkien[a]

[a]*Eindhoven University of Technology, Den Dolech 2, 5600 MB Eidnhoven, The Netherlands*

Abstract

Reprogramming is an important issue in wireless sensor networks. It enables users to extend or correct functionality of a sensor network after deployment at a low cost. In this paper, we investigate the problem of improving energy efficiency and delay of reprogramming by using data compression and incremental updates. We analyze different algorithms for both approaches, as well as their combination, when applied to resource-constrained devices. Our results show that the classic Lempel-Ziv-77 compression algorithm with Bsdiff for delta encoding has the best overall performance compared to other compression algorithms; on average reducing energy usage by 74% and enabling 71% faster updates.

Keywords: Wireless Sensor Networks, Reprogramming, Data Compression, Delta Encoding, Incremental Update

1. Introduction

Wireless Sensor Networks are collections of interconnected autonomous sensor nodes. Nodes are cheap, small and resource constrained devices, powered by small batteries. Due to their low price and high deployment flexibility, they find applications in many areas.

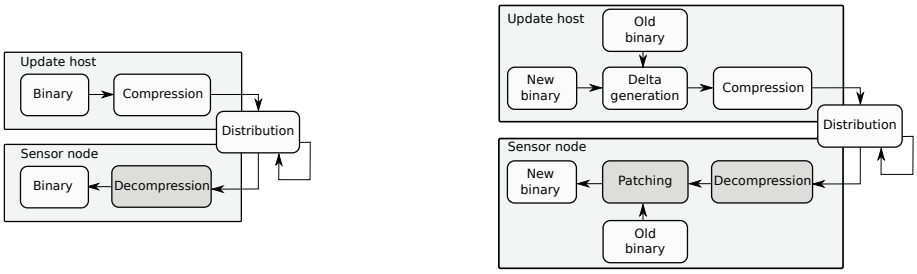
An important feature of wireless sensor networks is *reprogramming*, i.e. the capability to change software functionality of nodes within the network at run time. Changes come in the form of updates, consisting of new applications, bug fixes or modified parameters. Reprogramming is important both during development, for fast prototyping and debugging, and after deployment, for adapting functionality.

Due to the high number of nodes within a network and erroneous wireless media, reprogramming is a non-trivial task. As the network size increases, scalability issues such as delay, energy usage and reliability become crucial. Updates can be large, hence distributing them can take a long time. During the update process, the sensor network is unusable for other tasks. Finally, wireless transmission consumes valuable energy from node batteries, essentially reducing their life time.

Our research hypothesis is that reprogramming can be made more energy-efficient by compressing data before it is transmitted. Compression saves energy directly, by sending and receiving less data, and indirectly

Email addresses: m.stolikj@tue.nl (Milosh Stolikj), P.J.L.Cuijpers@tue.nl (Pieter J.L. Cuijpers), j.j.lukkien@tue.nl (Johan J. Lukkien)

1877-0509 © 2012 Published by Elsevier Ltd. Open access under CC BY-NC-ND license.
doi:10.1016/j.procs.2012.06.026



(a) Data compression

(b) Data compression and incremental update

Fig. 1. Overview of the update process when using only data compression or combined with incremental update.

by keeping the media free, thus lowering the number of retransmissions. Since the processor consumes much less energy than the wireless radio, additional processing is favourable compared to data transmission. Unfortunately, most compression algorithms rely on high memory usage and processing power to reduce the size of data, making them inappropriate in resource-constrained devices. We aim to find the best approach for compression on sensor nodes, considering the trade-off between performance and resource requirements.

There are two commonly used approaches for compression. The first one is to apply data compression directly to updates (Figure 1a). The second approach uses incremental updates (Figure 1b), and requires the transmission of only the difference between the two consecutive versions of an application. The difference, extracted in scripts called *deltas*, has redundant structure and can therefore be compressed to reduce size. Contrary to other applications of compression in sensor networks, during reprogramming, only decompression is needed on sensor nodes.

The contributions of this paper are in the analysis of the gains and pitfalls of using data compression and incremental updates in reprogramming wireless sensor networks. Even though both approaches have been studied before, we look at their unique combination, and their applicability in resource constrained devices. Among the variety of available algorithms, we search for the best possible combination. First, we systematically analyze the two approaches of data compression and incremental updates. Then, we establish metrics used for comparing different algorithms for both approaches. Finally, we combine the two and locate the best combination in terms of energy savings and delay. Our results show that 74% savings in energy usage can be achieved on average, with 71% less delay compared to transferring complete updates.

The paper is structured as follows. Section 2 covers related work on reprogramming of embedded systems. Section 3 introduces common approaches to data compression and incremental updates. Section 4 describes the experiments done on sensors nodes. Finally, the results and conclusions are given in section 5.

2. Related work

Operating systems for resource-constrained devices usually do not have built in support for incremental updates. Non-modular systems can only be reprogrammed by replacing the entire firmware with a new one. Multiple mechanisms for disseminating firmwares and replacing them on nodes have been developed for both single-hop (XNP [1]) and multi-hop networks (Trickle [2], Deluge [3], Stream [4]). Incremental update is especially appealing when large firmware images are considered for dissemination. In [5], modified versions of the rsync [6] and XNP protocols are used for generating deltas and their dissemination, respectively. Zephyr [7] adds application-level modifications to decrease differences between consecutive application versions, then produces deltas with rsync and distributes them using Stream.

Modular operating systems such as Contiki [8] support dynamic linking and loading. They improve energy usage during reprogramming by using smaller partial executables for dissemination. For instance, Contiki uses the ELF format for holding partial executables, including symbol and relocation tables. However, these tables make ELF binaries potentially large for transfer. In our work we use binary images and

ELF executables for the Contiki operating system as test data, but due to the general nature of the algorithms used, the results are applicable to any other architecture.

Compression has been previously considered in sensor networks, mostly for data gathered from sensors. In [9], several algorithms are compared on desktop machines, for compressing data from two test beds. Similarly, in [10] compression algorithms are compared on ELF executables for the Contiki operating system. However, during upgrades, only decompression is needed on resource-constrained devices. It is presently unclear how much resources are needed to add only decompression. Furthermore, previous studies do not consider combining incremental updates and compression algorithms, which is explored in this work.

3. Methodology

There are two common approaches for reducing the size of data in software updates: using data compression and incremental updates. Next, we discuss both approaches individually.

3.1. Performing updates using data compression

Compression, and accordingly decompression, is added to the update process as shown in Figure 1a. It is an intermediate phase for representing information with fewer bits.

Compression algorithms can be categorized by their theoretical foundations. Often this foundation influences the amount of resources they use and the maximum achievable compression ratio. Since we are working with executable data, where every bit is important, we consider only lossless algorithms. Furthermore, compression is done outside the sensor network, so only decompression is needed on sensor nodes.

Entropy-based algorithms, such as Shannon-Fano coding and Huffman coding [11], find optimal representation of symbols found in uncompressed data. They establish a prefix-free code for each symbol in the input. Then, every fixed-length symbol is replaced by a variable-length prefix-free code word. Decompression is the opposite process: each code word is replaced by a fixed-length symbol. Entropy encoders are usually slow and require significant memory to store prefix lists, but produce small compressed data.

Dictionary based compression algorithms, or Lempel-Ziv (LZ) [12] variants, maintain a look-up dictionary of frequent symbols sequences [13] [14] [15]. Whenever a match is found in the uncompressed data, it is replaced with a reference to the dictionary. The dictionary can be simply the previous symbol (Run-length encoding, RLE) or a sliding window of the previously processed data.

In order to improve the compression ratio, data can be pre-processed. By re-arranging uncompressed data, a more compressible representation can be achieved. We use one such algorithm, BZip2 [16], as a reference point for the approximate maximum compression ratio, although it can not be run on resource-constrained devices. Other means pre-processing include use of additional data, such as incremental updates.

3.2. Performing updates incrementally

Most changes in software come in the form of incremental updates, which either add additional functionality or modify values of existing parameters. The *old* and *new* version share most of the code base, and the difference between them is usually several times smaller than the size of the application itself.

Algorithms for delta encoding exploit this behaviour by extracting and distributing only the differences between both versions. The delta contains instructions and data, which are used to reconstruct the new version from the old one, a process called patching. Delta encoding algorithms differ in how the delta is constructed and how the differences are detected.

We have selected Bsdiff [17] as a favourable delta encoding algorithm. Previous research [18] showed that on average, it provides smallest deltas compared to other algorithms. It builds deltas in two passes. In the first pass, completely identical blocks are found in both versions. Next, it tries to expand exact matches in both directions, such that every prefix/suffix of the extension matches in at least half of its bytes. The delta is then constructed of three parts: a control block of commands; a diff block of bytewise differences between approximate matches; and an extra block of new data. When there are large similarities between the old and new version, the diff block contains large series of zeroes, which can be easily compressed.

All delta encoding algorithms use external compression algorithms to reduce the delta's size. Therefore, by adding delta encoding, a sensor node is reprogrammed as in Figure 1b. Delta encoding can be seen as a pre-processor; it is an initial phase that improves the performance of data compression algorithms.

4. Evaluation

This section describes the metrics we use for our tests, the hardware on which the tests are performed, and the results for each metric.

4.1. Metrics

For compression algorithms on resource-constrained devices, four metrics are relevant: code size of the algorithm, memory used during execution, energy and delay. The size of compressed data and number of processor cycles are two additional factors which directly determine energy and delay savings.

The reduction in size of the compressed data is quantified through the compression ratio. It is a standard metric used to compare compression algorithms, defined as the reduction in size relative to the uncompressed data: $compr\_ratio = (1 - \frac{compressed\_size}{uncompressed\_size}) * 100$. Therefore, higher values mean smaller compressed files, hence better performance.

On the other hand, decompressing data requires a certain amount of processor cycles. Since sensor nodes have low frequency processors, a high number of processor instructions would result in large decompression times. Therefore, this value should be as low as possible.

Memory is limited in resource-constrained devices. This includes both memory required for holding the code, which is stored in internal flash memory (ROM), and memory required during execution, in RAM. Algorithms running on sensor nodes must have a small code footprint, up to a couple of kilobytes, and use little memory during execution.

We estimate energy usage through a linear model which relies on the amount of time spent during computation and transmission of data [19]. This is a lower bound of the real energy usage; we assume that forwarding is done immediately, without additional processing, and we ignore MAC protocol behavior. Adding those variables, especially the influence of a low duty-cycle MAC protocol, will result in higher energy usage for transmission, penalising communication even further. In general, we estimate energy usage for reprogramming a node within a network of h neighbours as:

$$E = (k_{err} * E_{recv} * n_{packets}) + (k_{err} * h * E_{send} * n_{packets}) + E_{cpu},$$

where k_{err} is the average number of times each packet is sent, the number of sent/received packets is $n_{packets} = \lceil \frac{data\_size}{payload\_size} \rceil$, $payload\_size$ is the maximum amount of data that can be fit in one data frame, E_{recv} and E_{send} are the energy required to receive/send one packet and E_{cpu} is the energy required for post-processing of the received data. Transmission energy is expressed as $E_{send/recv} = t_{send/recv} * I_{send/recv} * V$, where $t_{send/recv}$ is the amount of time that the wireless radio is in sending/listening state, $I_{send/recv}$ is the current, and V is the voltage. Similarly, $E_{cpu} = I_{cpu} * V * \frac{CPU_{cycles}}{CPU_{freq}}$, where CPU_{cycles} is the number of processor instructions, and CPU_{freq} is the frequency of the processor. In the most basic case, where no post-processing is used, E_{cpu} can be ignored. In the second case, when compressed data is received, E_{cpu} accounts for decompression energy. Finally, in the third case, when compressed deltas are received, E_{cpu} captures energy used for both decompressing and patching.

We estimate the time needed to complete an update with a similar model to the one used for energy estimation. Again we estimate a lower bound of the delay, since we assume that forwarding is done immediately, and that the MAC protocol does not introduce additional overhead:

$$t = (k_{err} * t_{recv} * n_{packets}) + (k_{err} * h * t_{send} * n_{packets}) + \frac{CPU_{cycles}}{CPU_{freq}}.$$

In most cases, the energy model and delay model give similar results. The difference between them comes in the scaling factors added in the energy model for expressing appropriate energy usage. Therefore, these two metrics behave differently when for two measurements:

$$\frac{I_{cpu}}{k_{err} * CPU_{freq} * (I_{recv} * t_{recv} + h * I_{send} * t_{send})} < \frac{n_{packets1} - n_{packets2}}{CPU_{cycles2} - CPU_{cycles1}} < \frac{1}{k_{err} * CPU_{freq} * (t_{recv} + h * t_{send})}.$$

4.2. Experimental setup

To verify the effect of compression and delta encoding in reprogramming wireless sensor networks, we considered four scenarios for reprogramming: 1) Version upgrade of the operating system; 2) Installation of a new application; 3) Version upgrade of an application, with large differences between versions; 4) Small patch of an application, i.e. parameter reconfiguration. Every scenario except the first one consists of two test cases: upgrading the system with a new firmware image and using partial executables (ELF) (Table 1).

For each test case, both the initial version and the new version are available. First, we compress the new version directly. Then, we produce an intermediate delta using Bsdiff, and apply compression to it. We measure all six metrics mentioned in the previous section only for decompression and patching, since data compression and delta creation is done outside of the sensor network.

In our experiments, we use the Contiki operating system, running on Crossbow TelosB nodes [20], with the Open Service Architecture for Sensors (OSAS) [21] application. The node contains an 8 MHz TI MSP430 microcontroller with the Chipcon CC2420 IEEE 802.15.4 radio transceiver. It has 48 KB program flash memory, 10 KB random access memory and 1 MB external flash.

While memory usage was measured on the sensor nodes, the number of processor cycles was measured on a desktop platform, and then was scaled down to correspond to the MSP430 microcontroller. Scaling factors were obtained on a per-algorithm basis, by executing the same algorithm on both platforms.

Table 1. Test scenarios and data size of firmware images and ELF executables.

Test	Description	Type	Starting size	Final size
1a	Contiki 2.3 → Contiki 2.4	Firmware	22.924	20.624
1b	Contiki 2.4 → Contiki 2.5	Firmware	20.624	22.980
2a	Contiki 2.5 + Hello world → Contiki 2.5 + OSAS 2.0	Firmware	22.980	39.112
2b	OSAS 2.0 (no previous version exists)	ELF executable	-	26.712
3a	Contiki 2.5 + OSAS 1.0 → Contiki 2.5 + OSAS 2.0	Firmware	37.796	39.112
3b	OSAS 1.0 → OSAS 2.0	ELF executable	25.784	26.712
4a	Contiki 2.5 + OSAS 2.0 → Contiki 2.5 + OSAS 2.1	Firmware	39.112	39.112
4b	OSAS 2.0 → OSAS 2.1	ELF executable	26.712	26.712

4.3. Results

Next we will discuss each of the aforementioned metrics individually.

4.3.1. Compression ratio

Average compression ratio over all test cases, with and without incremental updates, is shown in Figure 2. It implies that incremental updates make significant difference in the performance of compression algorithms. Depending on the approach and type of updates that need to be compressed, between 35% and 99% compression ratio can be achieved.

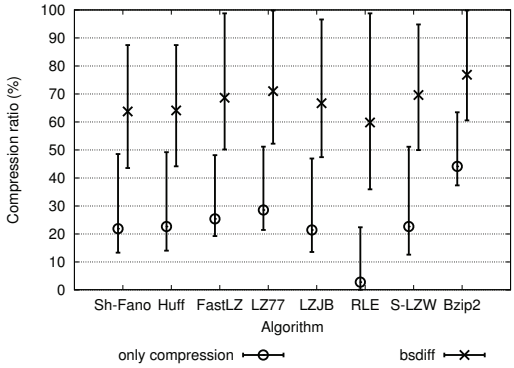


Fig. 2. Minimum, maximum and average compression ratio for the test cases in table 1. The left line (o) corresponds to compressing data directly, while the right line (x) corresponds to compressing the Bsdiff delta.

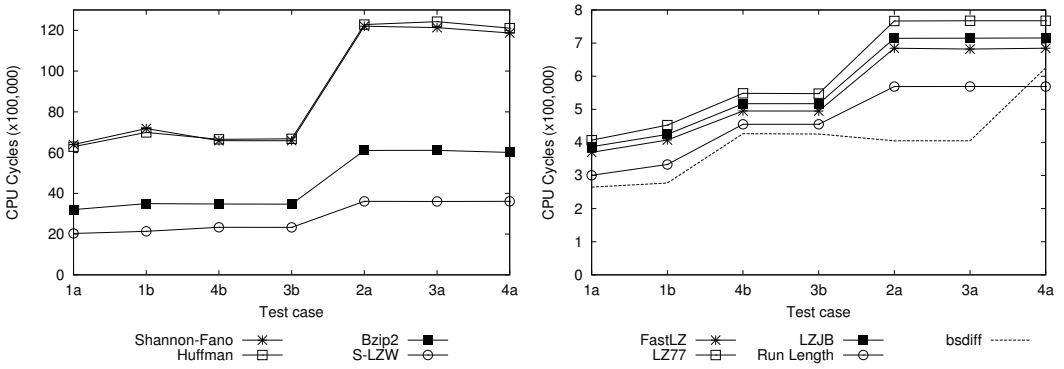
Most compression algorithms behave similarly, with not more than 10% difference between them. The two exceptions are RLE and BZip2, the worst and best compressor, respectively.

4.3.2. Number of processor instructions

Entropy based decompression algorithms, as shown in Figure 3a, require the most instructions. In fact, they are between 3 to 15 times more CPU intensive than the dictionary-based algorithms. This is a strong indication that such algorithms are inappropriate to use on resource-constrained with slow processors, and are therefore excluded from comparisons in the next three metrics.

This metric indicates that Lempel-Ziv variants are good candidates to be implemented on wireless sensor nodes. Surprisingly, Sensor-LZW consumes many processor instructions, which is not common for algorithms designed for sensor nodes.

The patching code of Bsdiff, compared to decompression algorithms, is lightweight in processing requirements. Figure 3b shows that it uses fewer processor instructions than any decompression algorithm in all except one test case.



(a) Slower algorithms (b) Faster algorithms

Fig. 3. Number of processor instructions for decompression and patching. Note the different scale on both charts.

4.3.3. Memory requirements

The memory resources required to add decompression and delta encoding support are shown in table 2.

RLE, LZ77 and LZJB are lightweight in terms of both code size and memory usage during execution. FastLZ has a significantly larger code base, but still uses little stack space. Sensor-LZW is resource-consuming in both categories.

Bsdiff also requires little RAM and ROM memory, hence adding it does not prove to be a burden.

Table 2. Code and memory footprint of different algorithms, disregarding buffer size. RAM refers to data memory and ROM refers to code memory.

Algorithm	ROM (bytes)	RAM (bytes)
FastLZ	1.484	52
LZ77	370	42
LZJB	302	38
RLE	166	24
Sensor-LZW	1.236	130
Bsdiff	370	48

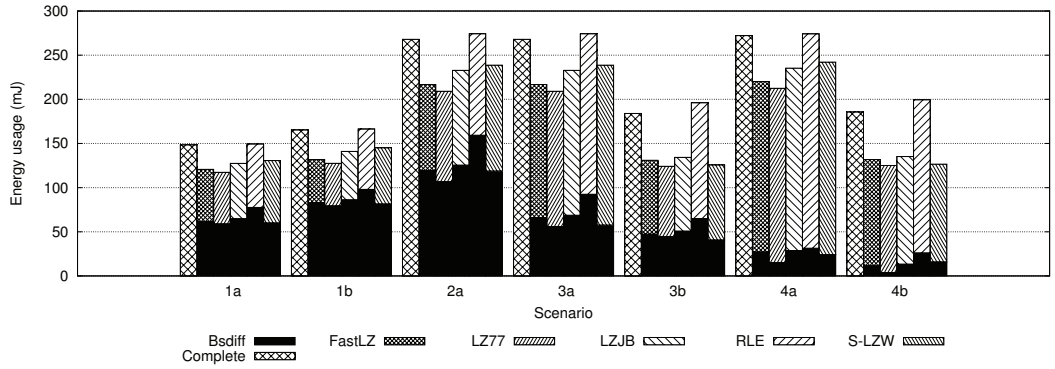


Fig. 4. Energy usage for reprogramming one node for each scenario. In every scenario, the leftmost bar is energy consumption without using decompression or patching. Each other bar corresponds to one compression algorithm and holds two values - energy usage when using only decompression (pattern segment) and energy usage when using both decompression and patching (black segment).

4.3.4. Energy estimation

Figure 4 shows the three energy estimations in an ideal channel with no loss ($k_{err} = 1$), for updating one TelosB node ($h = 1$), using timing estimations from [22].

Using only compression does not drastically reduce energy usage, ranging between 11% and 32%. RLE actually requires more energy, since it is not able to compress binary data at all, but further requires energy for decompression. Highest energy savings are achieved using LZ77.

Incremental updates significantly decreases energy usage for every compression algorithm considered. Reductions vary between 40% and 97.9%. The penalty for lower compression ratio is particularly evident for RLE, since it provides less energy savings than any of the other compression algorithms. The best performer is again LZ77, slightly better than the other three algorithms.

4.3.5. Delay

As shown in Figure 5, delay behaves similarly to the previous metric. LZ77 again has highest savings in delay, both when using only decompression or in combination with patching. In the first case, it reduces time between 19% between 31%. Sensor-LZW is heavily penalised for its high number of processor instructions, and in five of the test cases when using only decompression, it reduces delay by less than one percent. RLE actually makes things worse when used without incremental updates.

When using incremental updates, LZ77 reduces the time needed for a complete update of one node between 49% and 95%. Both Sensor-LZW and RLE perform much better in this case, reducing delay between 38%-86% and 41%-85%, respectively. In some cases, as in test 4a, RLE requires less time than Sensor-LZW, even though it uses more energy for update.

5. Discussion and conclusion

In this paper we have investigated two approaches for efficient reprogramming of wireless sensor networks. First, we evaluated the performance of data compression algorithms applied directly on binary data. Second, we combined Bsdiff, an algorithm for delta encoding, with the previously analyzed compression algorithms. Based on results on our experimental data, we selected five Lempel-Ziv variants for data compression for deployment on sensor nodes. We completed further tests to measure memory requirements and code footprint of all algorithms, as well as execution time, energy usage and delay for performing updates.

The presented results show that data compression and incremental updates improve reprogramming of wireless sensor networks in terms of energy efficiency and time required for update. Improvements vary depending on the selection of specific algorithms.

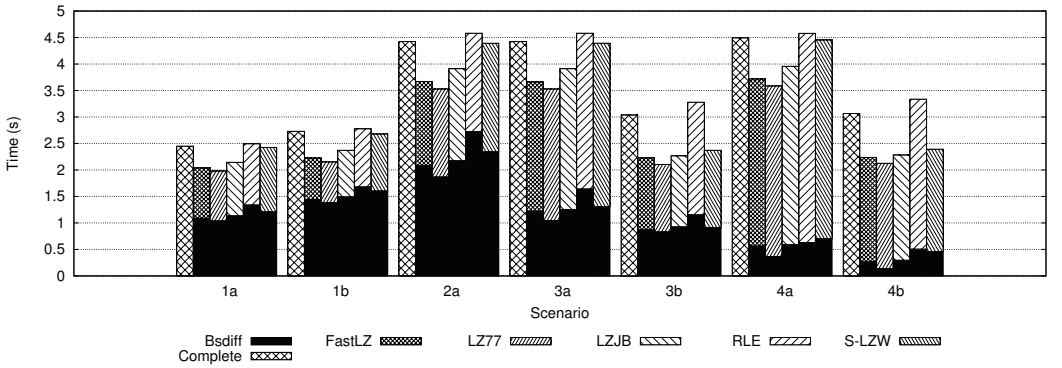


Fig. 5. Delay for reprogramming one node for each scenario. In every scenario, the leftmost bar is delay without using decompression or patching. Each other bar corresponds to one compression algorithm and holds two values - delay when using only decompression (pattern segment) and delay when using both decompression and patching (black segment).

We demonstrated that simply adding compression does not always lead to lower energy usage or faster updates. In fact, as shown in section 4, if not done properly, it can degrade performance. Contrary, incremental updates showed a consistent improvement in all test cases. Since the entropy of generated deltas is relatively low, compression algorithms produce significantly smaller data. A strong argument for using incremental updates is that the minimum amount of measured energy savings is 40%.

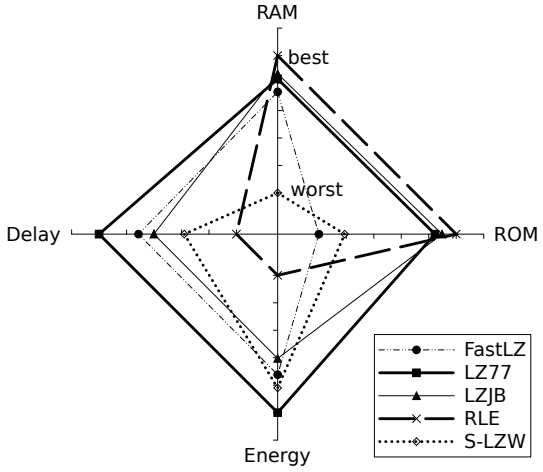


Fig. 6. Overall comparison of compression algorithms used in combination with Bsdiff. Average values for all metrics are scaled and ordered, from worst to best.

Selecting the best compression algorithm to use in combination with incremental updates depends on available resources. We have summarized all measured metrics for the data compression algorithms in Figure 6. It gives general directions for which algorithm to select, depending on requirements. For instance, if low memory usage is a necessity, then RLE would be the best choice. Overall, we have identified LZ77 as the algorithm with the finest balance between performance and resource requirements. It provides largest energy and delay savings, combined with modest RAM and ROM requirements.

The energy and delay models can be made more precise by including timing needed to transfer data from and to flash memory for large updates. On architectures like the Crossbow TelosB, where flash memory is

accessed through the same bus as the radio chipset, issues may arise with synchronizing memory access with the radio duty cycle. In such cases CPU intensive algorithms as Sensor-LZW might be inappropriate.

Some compression algorithms, e. g. Sensor-LZW, can be configured with various options which have influence on execution time, memory requirements and compression ratio. We leave the investigation of most appropriate parameter values of such algorithms for specific platforms as future work.

Acknowledgement

The authors would like to thank Martijn van den Heuvel and Richard Verhoeven for their valuable discussions and improvements on this article. This work is supported in part by the Dutch P08 SenSafety Project, as part of the COMMIT program.

References

[1] I. Crossbow Technology, Mote in-network programming user reference version 20030315 (2003).
[2] P. Levis, N. Patel, D. Culler, S. Shenker, Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks, in: Proc. Symp. on Networked Systems Design and Implementation - Volume 1, USENIX, 2004, pp. 2–2. URL http://dl.acm.org/citation.cfm?id=1251175.1251177
[3] J. W. Hui, D. Culler, The dynamic behavior of a data dissemination protocol for network programming at scale, in: Proc. Int. Conf. on Embedded networked sensor systems, SenSys, ACM, 2004, pp. 81–94. doi:10.1145/1031495.1031506.
[4] R. Panta, I. Khalil, S. Bagchi, Stream: Low overhead wireless reprogramming for sensor networks, in: Proc. Int. Conf. on Computer Communications, INFOCOM, 2007, pp. 928 –936. doi:10.1109/INFCOM.2007.113.
[5] J. Jeong, D. Culler, Incremental network programming for wireless sensors, in: Conf. on Sensor and Ad Hoc Communications and Networks, IEEE SECON, 2004, pp. 25 – 33. doi:10.1109/SAHCN.2004.1381899.
[6] A. Tridgell, Efficient algorithms for sorting and synchronization (2000).
[7] R. K. Panta, S. Bagchi, S. P. Midkiff, Efficient incremental code update for sensor networks, ACM Trans. on Sensor Networks 7 (2011) 30:1–30:32. doi:10.1145/1921621.1921624.
[8] A. Dunkels, B. Grnvall, T. Voigt, Contiki - a lightweight and flexible operating system for tiny networked sensors, in: Proc. IEEE Workshop on Embedded Networked Sensors (Emnets-I), 2004. doi:10.1109/LCN.2004.38.
[9] K. Dolfus, T. Braun, An evaluation of compression schemes for wireless networks, in: Int. Cong. on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT), 2010, pp. 1183 –1188. doi:10.1109/ICUMT.2010.5676532.
[10] N. Tsiftes, A. Dunkels, T. Voigt, Efficient sensor network reprogramming through compression of executable modules, in: Conf. on Sensor, Mesh and Ad Hoc Communications and Networks, SECON, 2008, pp. 359 –367. doi:10.1109/SAHCN.2008.51.
[11] M. Nelson, The Data Compression Book, Henry Holt and Co., Inc., New York, NY, USA, 1991.
[12] J. Ziv, A. Lempel, A universal algorithm for sequential data compression, IEEE Trans. on Information Theory 23 (3) (1977) 337 – 343. doi:10.1109/TIT.1977.1055714.
[13] C. M. Sadler, M. Martonosi, Data compression algorithms for energy-constrained devices in delay tolerant networks, in: Proc. Int. Conf. on Embedded networked sensor systems, SenSys, 2006, pp. 265–278. doi:10.1145/1182807.1182834.
[14] A. Hidayat, Fastlz, free, open-source, portable real-time compression library. URL http://www.fastlz.org/
[15] J. Bonwick, Lzjb compression algorithm.
[16] J. Seward, A program and library for data compression. bzip2 and libbzip2. URL http://www.bzip.org
[17] C. Percival, Naive differences of executable code (2003). URL http://www.daemonology.net/bsdiff/
[18] M. Stolikj, P. J. L. Cuijpers, J. J. Lukkien, Efficient reprogramming of sensor networks using incremental updates and data compression, Tech. Rep. CS-12-10, Eindhoven University of Technology (2012).
[19] D. Albu, J. Lukkien, R. Verhoeven, Energy effect of on-node processing of ecg signals, in: Consumer Electronics (ICCE), 2010 Digest of Technical Papers Int. Conf., 2010, pp. 7 –8. doi:10.1109/ICCE.2010.5418748.
[20] J. Polastre, R. Szewczyk, D. Culler, Telos: enabling ultra-low power wireless research, in: Proc. Int. Symp. on Information processing in sensor networks, IPSN, IEEE Press, 2005. doi:10.1109/IPSN.2005.1440950.
[21] R. Bosman, J. Lukkien, R. Verhoeven, An integral approach to programming sensor networks, in: 6th IEEE Consumer Communications and Networking Conf., CCNC, 2009, pp. 1 –5. doi:10.1109/CCNC.2009.4784846.
[22] T. R. Burchfield, S. Venkatesan, D. Weiner, Maximizing throughput in zigbee wireless networks through analysis, simulations and implementations (2007).