Efficient reprogramming of sensor networks using incremental updates and data compression

Technische Universiteit Eindhoven
Department of Mathematics and Computer Science

Efficient reprogramming of sensor networks using incremental updates and data compression

Milosh Stolikj, Pieter J.L. Cuijpers, Johan J. Lukkien

12/10

ISSN 0926-4515

editors: prof.dr. P.M.E. De Bra
prof.dr.ir. J.J. van Wijk

Reports are available at:
http://library.tue.nl/catalog/TUEPublication.csp?Language=dut&Type=ComputerScienceReports&Sort=Author&level=1 and
http://library.tue.nl/catalog/TUEPublication.csp?Language=dut&Type=ComputerScienceReports&Sort=Year&Level=1

Computer Science Reports 12-10
Eindhoven, June 2012

Efficient reprogramming of wireless sensor networks using incremental updates and data compression

Milosh Stolikj, Pieter J. L. Cuijpers, Johan J. Lukkien
Eindhoven University of Technology
Department of Mathematics and Computer Science
System Architecture and Networking Group
{m.stolikj,p.j.l.cuijpers,j.j.lukkien}@tue.nl

May 2012

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 two problems: improving the energy efficiency and improving the delay of reprogramming. As enabling technologies we use data compression and incremental updates. We analyze different algorithms for both approaches, as well as their combination, when applied to resource-constrained devices. All algorithms are ported to the Contiki embedded operating system, and profiled for different types of reprogramming. Our results show that there is a clear trade-off between performance and resource requirements. Either VCDIFF, or the combination of Lempel-Ziv-77 or FastLZ compression algorithms with BSDIFF for delta encoding, have the best overall performance compared to other compression algorithms.

1 Introduction

Wireless Sensor Networks consist of interconnected autonomous sensor nodes placed over an area of interest, with high density. Due to their low price and high deployment flexibility, they are more and more used in various applications such as improving public safety [1], disaster management [2], environmental [3], structural [4] and traffic monitoring [5] etc.

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.

We can categorize reprogramming according to the type of change that is required in the network and on the nodes themselves. In general, we call these modifications *updates*. We can distinguish:

- an update of the operating system;
- an update of an application;
- an addition of a new application;
- a modification of parameters in an existing application.

Wireless sensor nodes are usually reprogrammed in two ways: either by flashing the node with a complete firmware image, or by loading a partial executable binary. The second approach is more flexible and allows easier extension of applications, without the need to reboot the operating system. Despite its flexibility, it has limited support on existing sensor network platforms. However, the Contiki operating system [6], which is used in this report, is specifically designed for wireless sensor networks and has dynamic linking as a core functionality.

Updates are assembled at a host machine outside the sensor network, and they are initially injected in the network through a gateway node. From then on, the update is spread within the network through some propagation mechanism. In order to improve propagation time, a selected number of nodes, or even all nodes that have received an update, forward it further to other nodes within their ranges (Figure 1).

Both firmware images and partial executables can be rather large. For instance, an image for the TelosB motes [7] with the Contiki operating system is around 23 KB in size. Adding additional applications can increase the size of the binary up to 48 KB. In order to transfer this image, between 200 and 430 IEEE 802.15.4 data frames must be used. Similarly, a partial executable (ELF) of an application can be 25 KB in size, and transferring it would require around 220 data frames.

Due to the high number of nodes within a network, the size of the data that needs to be sent, limited capacity of nodes, and the 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. 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.

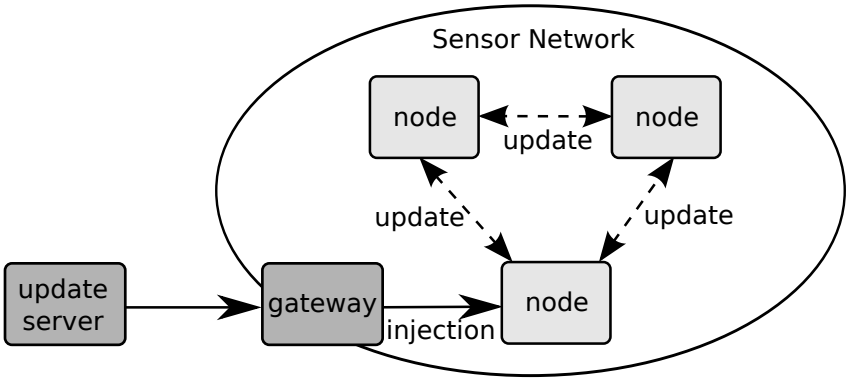


Figure 1: Propagation of updates in a wireless sensor network.

Our research hypothesis is that reprogramming wireless sensor nodes can be made more energy-efficient by compressing data before it is transmitted, and we want to investigate what is the best possible method for compression. Using compression energy is saved directly, by sending and receiving less data, and indirectly by keeping the media free, thus reducing the chances for collisions to occur. Since the processor requires 10 times less energy than the wireless radio for transmission, the additional processing for decompression is favourable to wireless transmission.

Two approaches to compression are commonly used. The first one is to apply data compression algorithms directly to the updates (binary data). The second approach uses the fact that when an application is updated, much of the old and new version remains the same. Therefore, by using incremental updates, only the difference between the two would need to be known and transmitted. This difference is expressed in scripts called *deltas*, and the according algorithms are known as delta encoding. Deltas have a highly compressible structure and can actually improve the performance of the data compression algorithms. A comforting fact is that since the data is compressed and encoded outside of the sensor network, a sensor node only has to be able to decompress it and apply the delta.

Unfortunately, most compression and delta encoding algorithms are not designed for the scarce resources of wireless sensor nodes. For instance, the Crossbow TelosB has a 8MHz microcontroller with 10kB RAM, 48kB program flash memory and 1MB serial flash, which makes it difficult to fit multiple applications in the same memory space.

An ideal compression and delta encoding algorithm for sensor nodes should have the following properties:

- Minimal energy usage,
- Minimal time to complete the reprogramming (delay),
- Minimal processing requirements,
- Small memory footprint (in ROM),

- Small RAM usage.

The first two properties are necessary for making the entire compression step feasible - unless the algorithm is able to considerably reduce the energy and time required for completing the update process, there is no point in using it at all. Due to the high energy requirements for transmitting data over the air, these properties are directly influenced by compression ratio, i.e. the ability to reduce the size of data as much as possible.

The next three properties are more subtle, and come from the nature of the wireless nodes themselves. As previously stated, they have limited resources including a slow processor, so algorithms running on them have to be efficient or simple, in order to be used in real time. Additionally, compression is a *support* feature and not the core functionality of the system, so it has to occupy as little space as possible. Small memory footprint is important during run-time, to have enough room to fit other applications in ROM and RAM memory. Furthermore, if reprogramming is a sporadic task, the small memory footprint would allow the code for decompression to be sent along with the update, so that it can be run only when needed.

An algorithm that optimally satisfies all requirements is not available. Therefore choosing one would involve a trade off between performance (i.e. compression ratio) and resource requirements. The aim of this report is also to show what are the gains and pitfalls of using data compression and incremental updates in reprogramming wireless sensor networks.

Our contributions are three fold. First, we systematically analyze different types of software update in sensor networks. Second, we benchmark how much resources are required to add support for compression and delta encoding on sensor nodes. Finally, based on pre-defined requirements, we provide a decision tree to find the best way to complete an update.

For our analysis, we selected seven data compression and three delta encoding algorithms. Then, we used the two most promising delta encoding algorithms, along with the feasible data compression algorithms, in four experiments for different update types, ranging from upgrade of the operating system to modifying a parameter in an application. Our results show that using incremental update in combination with data compression can reduce the size of updates up to 99%. Even simple techniques like run length encoding can achieve 40% reduction in size of updates. The best combinations yielded up to 95% savings in energy usage and delay.

The paper is structured as follows. Section 2 covers related work on compression and reprogramming in embedded systems. Section 3 introduces common approaches to data compression and incremental updates. Section 4 reports experimental results on combining compression and binary delta encoding algorithms on sensors nodes. The results are discussed in section 5, along with ideas for future work. Finally, section 6 gives conclusions and summary of the work presented in this paper.

2 Related work

Reprogramming has been an important area of interest in wireless sensor networks. Multiple solutions have been developed for it, ranging from extensions for operating systems for supporting remote reconfiguration, to the design of modular operating systems with features for dynamic linking and partial executables.

Modular operating systems such as Contiki [6], LiteOS [8] and RETOS [9] support dynamic linking and loading. Contiki uses the ELF format for holding partial executables, including symbol and relocation tables. This makes ELF binaries potentially large for transfer in noisy wireless sensor networks. In our work we use firmware images and ELF executables for the Contiki operating system as test data, but due to the general purpose nature of the algorithms used, the results are applicable to any other architecture as well.

On the other hand, non-modular systems such as TinyOS [10] 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 [11]) and multi-hop networks (Trickle [12], Deluge [13], MOAP [14], MNP [15], Stream [16], Cascade [17]). The same are often used by modular operating systems for dissemination of partial executables.

Incremental update is especially appealing when large firmware images are considered for dissemination. In [18], modified versions of the rsync and XNP protocols are used for generating deltas and their dissemination, respectively. Zephyr [19] adds application-level modifications to decrease the difference between consecutive application versions, then produces deltas with *rsync* and distributes them using

Stream. In [20], a tool similar to the *UNIX diff* is used to create deltas between versions. It extends the delta functionality with two new instructions, named repair and patch, which enable more efficient coding of the differences.

Considerable effort is directed towards expanding non-modular operating systems with dynamic linking capabilities. For instance, TOSThreads [21] is a library for TinyOS which adds thread support as well as dynamic linking of new services. Flexcup [22] takes a more general approach, aiming to provide on the fly reinstall of software components without the help of the operating system, by using symbol and relocation tables for every binary component. These tables are distributed along with the executable code to nodes, where re-linking and address binding is done. As with ELF executables, symbol and relocation tables can grow large, thus producing large updates.

Alternative methods such as virtual machines and middle-ware layers (Maté [23], OSAS [24]) overcome limitations of large updates for distribution by running interpreted code. Since byte code is much smaller compared to compiled binary code, updates in these systems can be easily distributed. The downside of this approach is that interpreted execution is slower and some resources are always used by the virtual machine. Still, the same problem is present if the operating system or the virtual machine engine need to be updated.

Compression has been previously considered in sensor networks, mostly for data gathered from sensors. In [25], several algorithms are compared on desktop machines, for compressing data from two test beds. Similarly, in [26] 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

3.1 Performing updates using data compression

Compression, and accordingly decompression, is added to the update process as shown in figure 2. It is an intermediate phase with the aim to encode information with fewer bits than the original representation. Many algorithms for it have been developed, ranging from task-specific to general purpose. Most general purpose compression algorithms are intended to be run on desktop systems, utilizing resources which are not available in resource-constrained devices. Therefore, even though some algorithms can compress data well, they may be inapplicable to wireless sensor nodes.

Compression algorithms can be categorized according to their theoretical foundations. Often this influences the amount of resources they use, as well as the maximum achievable compression ratio.

Since we are working with executable data, where every bit is equally important, we focus only on lossless algorithms. Furthermore, compression is done on powerful systems, usually an update host outside of the sensor network, so only decompression is needed on sensor nodes.

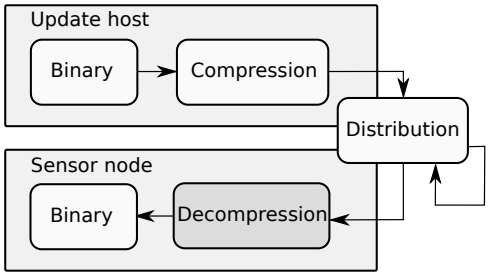


Figure 2: Overview of the update process when using data compression.

Entropy-based algorithms, such as Huffman coding [27], find optimal representation of symbols found in uncompressed data. They establish a prefix-free code for each symbol in the input, which assures that there is no code word that is a prefix of another code word. 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) [28] variants, maintain a look-up dictionary of frequent symbols sequences. 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.

Table 1: Comparison of families of compression algorithms.

Algorithm family	**Method**	**Resource usage**	**Tested algorithms**
Entropy-based	Optimal encoding	high	Huffman
Dictionary-based	Repeating sequences	moderate	LZ77, LZJB, FastLZ, RLE, Sensor-LZW [29–31]
Probabilistic, composite	Combination of several tecniques	extremly high	BZip2

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 [32], as a reference point for the approximate maximum compression ratio, although it can not be run on resource-constrained devices. Other means of 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. Similar to data compression, the delta creation is done outside of the sensor network, and only patching functionality needs to be added on the sensor nodes.

Most algorithms differ in how the delta is constructed and how the differences are detected.

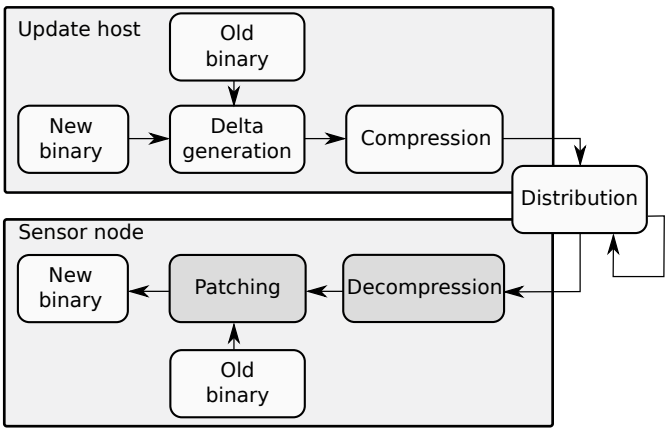


Figure 3: Overview of the update process when using incremental update and data compression.

Rsync, and the corresponding RDIFF algorithm [33], use non-overlapping fixed-sized blocks for matching identical data between the old and new version. Both versions are segmented into blocks, and for each one, a rolling-checksum and a MD5 checksum are computed. Based on these checksums, the delta is constructed of either references to blocks that already exist in the old version, or the entire content of new or changed blocks. While the rolling checksum is implemented to be as fast as possible, a MD5

checksum is definitely not appropriate for sensor nodes. An obvious weakness of the algorithm is that if two blocks differ in even one byte, the entire block content has to be present in the delta.

VCDIFF [34] is a format for encoding the difference between two data sets. The original idea for it comes from the Lempel-Ziv 77 algorithm - the old and new version are concatenated; then the resulting stream is compressed using LZ77 or a similar algorithm. From the output, the first part, which corresponds to the old version, is omitted, leaving only the instructions for the decoder to decompress the new version. VCDIFF features a detailed byte-code instruction set, consisting of a small number of instructions, which can be used in different addressing modes, accessing both the old and the new data. Additionally, a cache of recent addresses is held in memory. In this paper, we use Xdelta [35] as an encoder for generating VCDIFF deltas. It uses additional heuristics for optimizing the generated instruction set, such as removing completely covered instructions and combining small instructions into one, essentially reducing the delta size.

BSDIFF [36] is considerably different from the previous two approaches. It uses two passes to construct deltas. In the first pass, completely identical blocks are found in the two versions, using similar methods like the previous two. 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. These matches roughly correspond to modified lines of code. The delta is then constructed of three parts: a control block of commands for reconstructing the new version; diff block of bytewise differences between approximate matches and an extra block, consisting of new data. When there are large similarities between the old and new version, the diff block consists of large series of zeroes, which can be easily compressed.

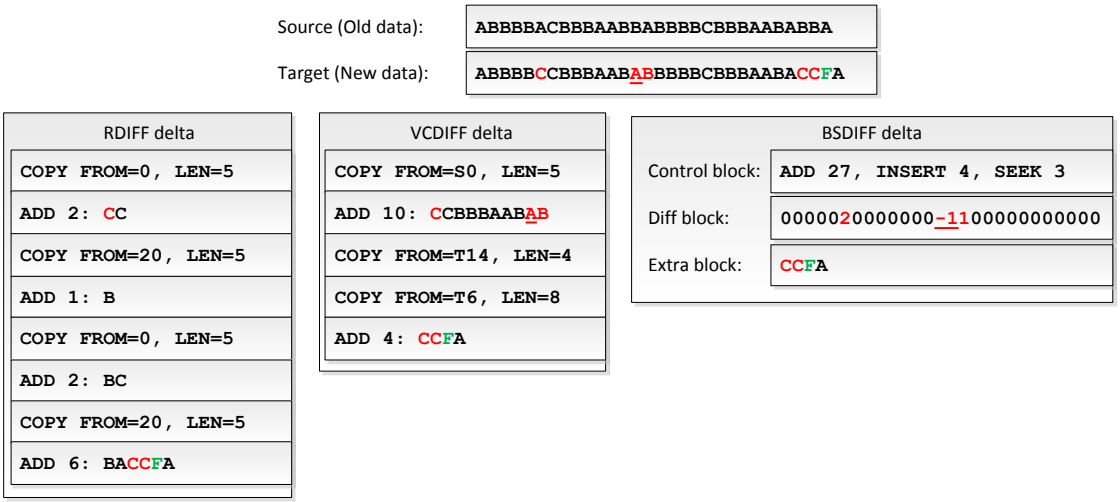


Figure 4: Comparison of the three delta types. For this example, RDIFF was set to use a blocks of 5 bytes. The RDIFF delta copies blocks of 5 bytes from the old data, everything in between is hardcoded with ADD instructions. The VCDIFF delta copies the first 5 bytes from the old data (S0), adds the next 10 bytes, then copies two block from the newly written data (T14 and T6). The last four bytes are again added from the delta. Finally, the ADD instruction in the BSDIFF delta specifies that the first 27 bytes from the old data and from the Diff block are summed. Zeroes in the Diff block mean that the corresponding byte from the old data is unchanged. The INSERT instruction adds four bytes in the Extra block to the output. The SEEK instruction moves the pointer in the old data to three places forward, to the end of the stream.

The behaviour of all three different delta encoding algorithms on a small sample is shown in figure 4. The RDIFF delta contains four COPY instructions for the corresponding 5 byte blocks from the input, everything in between is hardcoded in the delta. The VCDIFF delta specifies two types of source addresses for the COPY instruction - the first one references the old version, while the last two point to locations in the newly generated output. Again, everything that is not captured in the COPY instructions is hardcoded in the delta. Finally, BSDIFF has one ADD instruction, which performs binary addition between bytes in the diff block and the corresponding bytes from the old version. The last four bytes are

hardcoded, as part of the INSERT instruction.

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 3. Delta encoding can be seen as a pre-processor; it is an initial phase that improves the performance of data compression algorithms. An update host is responsible for producing deltas which need to be implemented on sensor nodes. A delta might consist of multiple patches; each patch is compressed separately and distributed to the node. There, after decompression, it is applied to the old version of the software. After all patches have been applied, the new version is complete and may be used.

Table 2: Comparison of algorithms for differential update.

Algorithm	**Matching type**	**Instruction set**
rdiff	identical blocks	small
vcdiff	identical blocks	complex
bsdiff	similar blocks	small

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 algorithms running 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 execution time are two additional factors which directly determine energy and delay usage.

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. A high number of processor instructions would result in large decompression times. Therefore, Regardless of processor speed, this value should be as low as possible. The importance will be captured in an overall model.

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 [37]. 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 in which each node has (at most) 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, $n_{packets} = \lceil \frac{data\_size}{payload\_size} \rceil$ is the number of sent/received packets, $payload\_size$ 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:

$$E_{send/recv} = t_{on} * I_{send/recv} * V$$

where t_{on} is the amount of time that the wireless radio is in sending/listening state. Previous research [38, 39] suggests that it takes 7ms to transmit a data frame. This includes time for performing the CSMA/CA assessment, transmission of the actual data and receipt of the acknowledgment. For simplification purposes, we assume that during reception, the radio chip is turned on for the same amount of time, and it draws more current. This corresponds to the values present in specifications of various radio chipsets, such as the CC2420.

Similarly, processing energy is calculated as:

$$E_{cpu} = I_{cpu} * V * t_{cpu}$$

where t_{cpu} is the amount of processing time.

In the most basic case, where no post-processing is used, E_{cpu} can be ignored:

$$E_{c1} = k_{err} * E_{recv} * \lceil \frac{uncompressed\_size}{payload\_size} \rceil + k_{err} * h * E_{send} * \lceil \frac{uncompressed\_size}{payload\_size} \rceil$$

In the second case, when only compression is used, the data needs to be decompressed after receipt. Therefore, the energy usage can now be estimated as:

$$E_{c2} = k_{err} * E_{recv} * \lceil \frac{compressed\_size}{payload\_size} \rceil + k_{err} * h * E_{send} * \lceil \frac{compressed\_size}{payload\_size} \rceil + E_{dcmp}$$

Finally, in the third case, when both incremental update and compression is used, both decompression and patching needs to be performed after receipt of the data. Assuming that the entire update is implemented through one patch, the energy usage can be estimated as:

$$E_{c3} = k_{err} * E_{recv} * \lceil \frac{compressed\_patch\_size}{payload\_size} \rceil + k_{err} * h * E_{send} * \lceil \frac{compressed\_patch\_size}{payload\_size} \rceil + E_{dcmp} + E_{patch}$$

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:

$$D = k_{err} * t_{recv} * n_{packets} + k_{err} * h * t_{send} * n_{packets} + t_{cpu}.$$

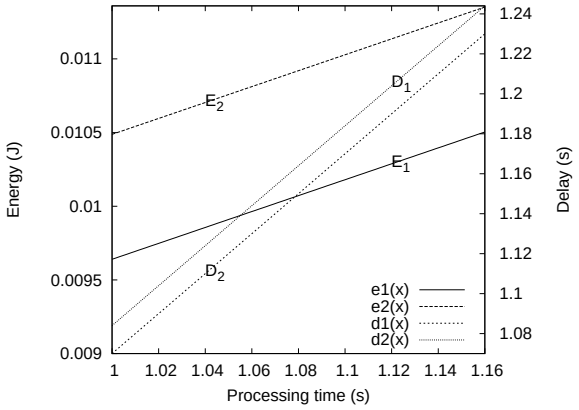


Figure 5: Example of two measurements where larger delay (D1¿D2) does not result in larger energy usage (E1¡E2). Calculated for $k_{err} = 1, h = 1, t_{send} = t_{recv} = 7ms, I_{proc} = 1.8mA, I_{send} = 17.4mA, I_{recv} = 23mA$. $e1(x)$ and $d1(x)$ correspond to energy usage and delay when $n_{packets} = 5$, while $e2(x)$ and $d2(x)$ use $n_{packets} = 6$.

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 energy usage. Therefore, these two metrics give contradicting results when for two measurements:

$$k_{err} * (t_{recv} + h * t_{send}) < \frac{t_{proc1} - t_{proc2}}{n_{packets2} - n_{packets1}} < \frac{k_{err} * (I_{recv} * t_{recv} + h * I_{send} * t_{send})}{I_{proc}}.$$

Figure 5 illustrates one such situation. It shows two curves for estimated energy usage ($e1$ and $e2$) and delay ($t1$ and $t2$), plotted against processing time. In the first case ($e1$ and $d1$), five data frames are used, compared to six data frames in the second case ($e2$ and $d2$). If we look at the energy usage and delay for two sample points, for instance at 1.04 and 1.12, we can notice that while energy usage is higher for the second case ($E_2 > E_1$), the delay has the opposite behaviour ($D_2 < D_1$).

4.2 Experimental setup and workflow

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 3).

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 each of the delta encoding algorithms, and apply compression to them. We measure the compression ratio of the compressed delta's with respect to the size of the new version. Then, we measure the other five 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 [7], with the Open Service Architecture for Sensors (OSAS) [24] 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.

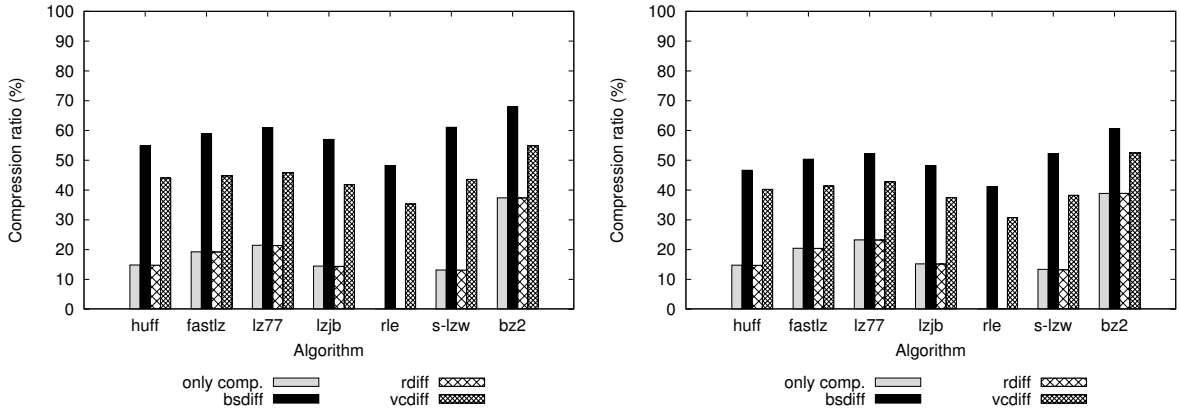
Both decompression and patching algorithms were adapted to be run on the TelosB nodes[1]. Input and output data is stored on the external serial flash and is accessed through the Contiki Coffee file system [40]. All tests were executed 10 times, and timed using the Contiki clock module.

Table 3: 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 is available)	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.

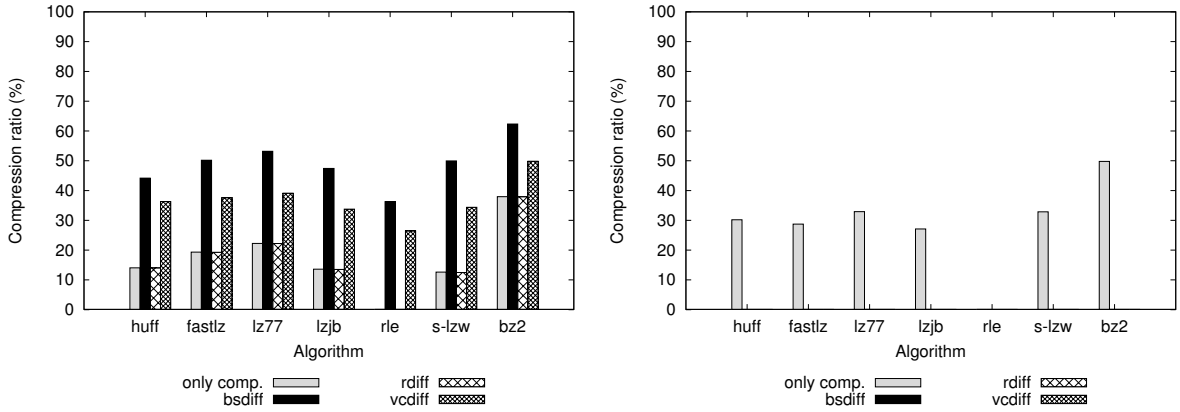


(a) Contiki 2.3 → 2.4

(b) Contiki 2.4 → 2.5

Figure 6: Compression ratio when upgrading the operating system (Test case 1).

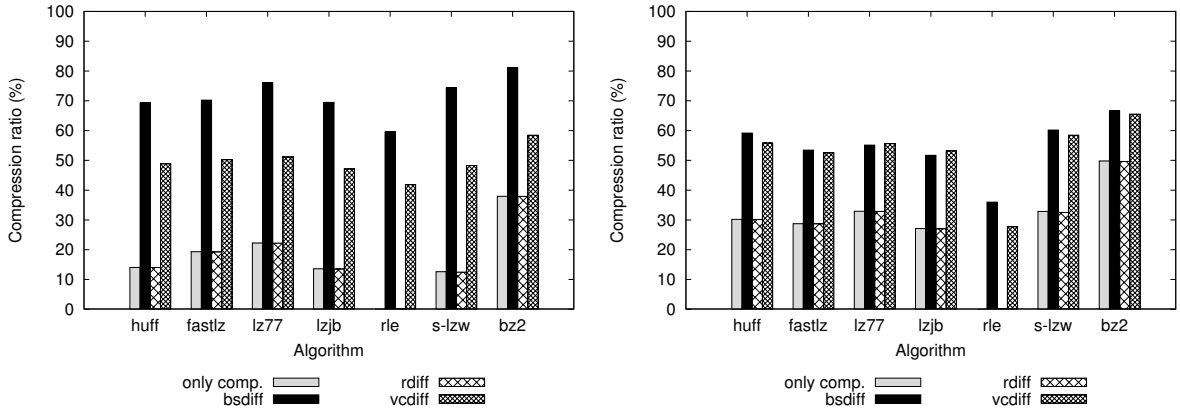
[1]The source code of the algorithms, as well as the data used in the report is available at `http://www.win.tue.nl/~mstolikj/compression/`. The port of VCDIFF to the MSP430 microcontroller was kindly provided by Nicolas Tsiftes.



(a) Firmware update

(b) Partial executable

Figure 7: Compression ratio when installing a new application (Test case 2).



(a) Firmware update

(b) Partial executable

Figure 8: Compression ratio when upgrading an application (Test case 3).

4.3.1 Compression ratio

Compression ratio is a factor which gives a strong indication what to expect from a compression algorithm in terms of energy and delay savings. As illustrated on figures 6, 7, 8 and 9, due to the rather diverse input samples, the compression ratio varies significantly between different test cases.

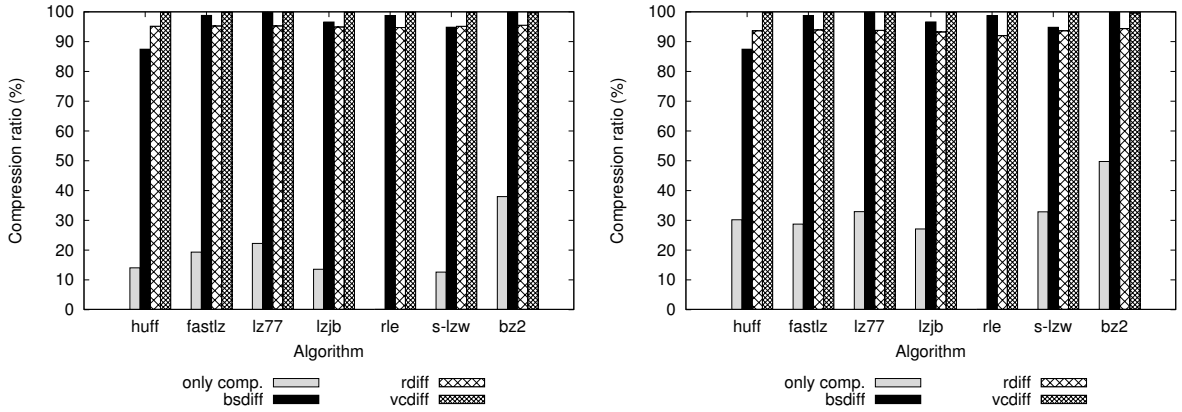
In the operating system update scenario, the compression ratio when using incremental update is approximately 10% higher in test case 1a (figure 6a) compared to test case 1b (figure 6b), while direct compression gives similar results. This is a clear indication that version 2.5 of the Contiki operating system is a major update to version 2.4, unlike 2.4 to 2.3.

The compression ratio in test case 2a (figure 7a) is lower than in the first one because the new firmware image is twice as large as the old one, hence a lot of data has not be seen before, and has to be inserted. With ELF executables, since there is no initial version of the application, delta encoding can not be used. Therefore, figure 7b shows only reported values by applying compression directly to the ELF executable.

The compression ratio shown in figure 8 is higher than in the previous two scenarios. The increase is obvious in both cases, when using firmware images and partial executables. The difference between the two versions is not as large as within the operating system, hence the deltas can be compressed better.

In the fourth scenario, the difference between the consecutive versions of the OSAS application is in two bytes, i.e. only the value of one integer variable is changed. Figure 9 shows that the compression ratio is very high, especially when using VCDIFF for delta encoding.

It is important to note that a parameter change may not always result in an assignment change in the generated machine code. As shown on figure 10, due to compiler optimizations, for nearly the same input, the compiler may generate completely different output. As a result, the old and new version will differ in more than two bytes, which generates significantly larger deltas.



(a) Firmware update

(b) Partial executable

Figure 9: Compression ratio when modifying a parameter (2 bytes) in an application (Test case 4).

Original source code

```
etimer_set(&et, CLOCK_SECOND * 600);
```

Generated assembly

```
5044: mov   #11264, r13
5048: mov   #1,     r14
504a: mov   #4542,  r15
504e: call  #0x69a4
```

New source code

```
etimer_set(&et, CLOCK_SECOND * 1800);
```

Generated assembly

```
5044: mov   #-31744,r13
5048: mov   #3,     r14
504c: mov   #4542,  r15
5050: call  #0x69a6
```

Figure 10: The compiler can generate completely different assembly/machine code for similar input.

In general, the compression ratio metric 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 37% and 99% compression ratio can be achieved.

Most compression algorithms behave similarly, with not more than 10% difference between them. The two exceptions are Run Length Encoding as the worst compressor and BZip2 as the best one.

Using BSDIFF showed higher compression ratio compared to the other delta encoding algorithms in all except the last test scenario. During the parameter change, VCDIFF produced smaller deltas. Disappointingly, RDIFF was inferior to the other two algorithms in all test cases, and was therefore omitted from the subsequent experiments.

4.4 Memory requirements

This metric determines the memory resources required to add decompression and delta encoding support. It can be divided in two parts - memory required for holding the code, which is stored in internal flash memory (ROM), and memory required during execution, in RAM.

Table 4 shows code and memory requirements for the six decompression algorithms, as well as the patching code of the VCDIFF and BSDIFF algorithms, ported to the Crossbow TelosB motes. The code size corresponds to the size of the .text segment of the ELF binary, while memory is the sum of static memory and maximum stack memory used during execution.

From the table, it is evident that Run Length Encoding, Lempel-Ziv 77 and LZJB are lightweight in terms of both code size and memory usage during execution; FastLZ has a larger code base, but still uses little stack space. The Huffman decoder has a small codebase, but uses a lot of memory to store the huffman tree, which contains 512 nodes. Finally, Sensor-LZW has the largest code base and uses the

Table 4: Code and memory footprint of different algorithms. All algorithms use a 2 byte buffer for holding data from/to serial flash memory.

Algorithm	Code (bytes)	Memory (bytes)
huffman	388	3388
fastlz	878	145
lz77	376	144
lzjb	424	140
rle	198	131
s-lzw	1.281	2502
bsdiff	560	158
vcdiff	2.261	1714

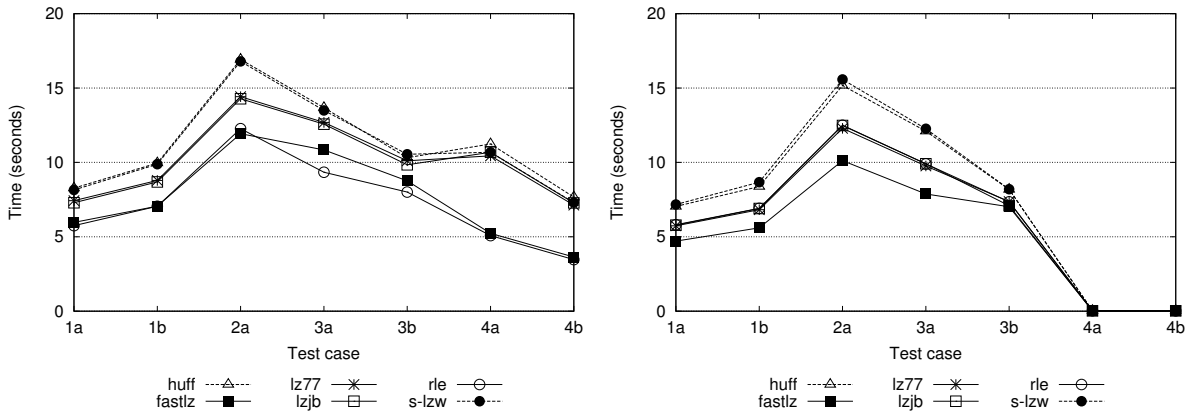
most memory of all decompression algorithms.

The memory footprint of BSDIFF is small, both in code size and memory usage. On the other hand, VCDIFF has a significantly larger code base, along with large memory footprint, mostly for storing the instruction cache.

4.5 Processing requirements

The time required to decompress the BSDIFF/VCDIFF deltas from the previous section is shown in figure 11. A buffer size of 2 bytes was used for all algorithms.

In all cases, Sensor-LZW and the Huffman decoder were the slowest algorithms. LZ77 and LZJB had almost identical execution times, while RLE had significantly worse performance while decompressing VCDIFF deltas. This comes down to the nature of the VCDIFF algorithm - run length encoding is added to the instruction set and is done while the delta is generated. Therefore, performing run length encoding on the delta does not give any improvement. Finally, on average, FastLZ was the fastest algorithm.



(a) Decompression time for a BSDIFF patch

(b) Decompression time for a VCDIFF patch

Figure 11: Time required for decompressing a patch.

Figure 11 shows that decompressing a VCDIFF delta is generally faster than decompressing a BSDIFF delta. The main reason for this visible behaviour is that the size of the VCDIFF delta is always smaller than the BSDIFF delta. This is especially expressed in test cases 4a and 4b, where the VCDIFF delta is around 60 bytes, while the BSDIFF delta is 39,130/26,724 bytes respectively. The decompression times become similar when the compressed BSDIFF delta is proportionally smaller than the compressed VCDIFF delta.

The time required to perform a patch is given in figure 12. As before, a buffer of 2 bytes was used.

Again, due to the size of the input data, the patching using VCDIFF is much faster than patching using BSDIFF.

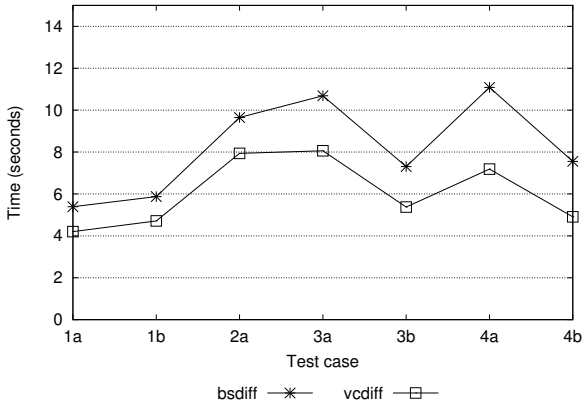
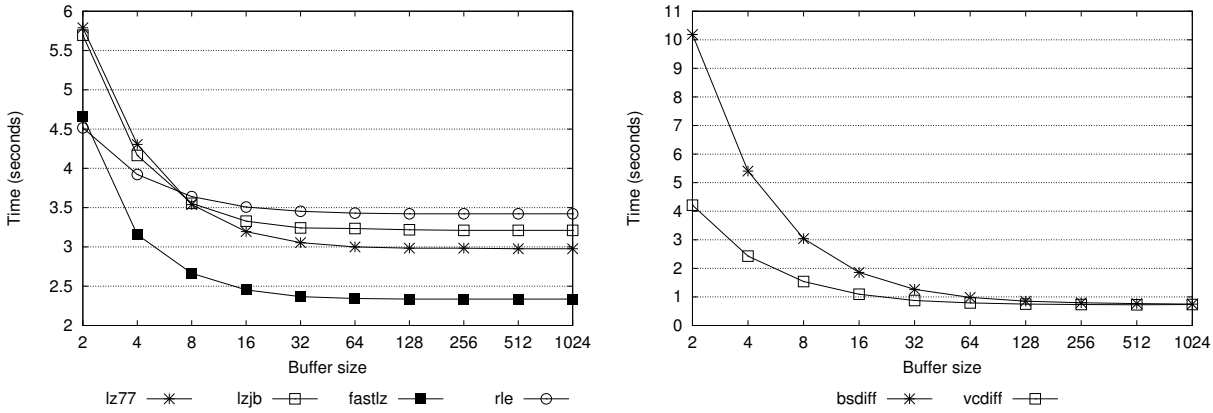


Figure 12: Time required for patching.



(a) Decompression time for the BSDIFF delta from test case 1a

(b) Patching time for test case 1a

Figure 13: Impact of buffer size on decompression and patching time.

A more detailed inspection of the buffer size gives significantly different insight. Figure 13a shows the behaviour of decompression time for one test case (decompressing a BSDIFF delta in test case 1a), with variable buffer size. From this experiment, Sensor-LZW and Huffman decoder were omitted, since they already use a lot of memory. Buffering was implemented only for block reads/writes; the input instructions are still read in sequences of 2 bytes. Buffer size plays an important role with FastLZ, LZ77 and LZJB - going from buffer size of 2 to 32 bytes gives almost 50% faster decompression. On the other hand, RLE performance gets saturated with buffer size 16 and higher, since it only uses the larger buffer when one symbol is written multiple times.

Similarly, figure 13b shows the impact of buffer size on patching time. The samples were computed for test case 1a. A larger buffer significantly improves patching time for BSDIFF, and with a buffer of 128 bytes, it is almost the same as VCDIFF. The improvements are higher with BSDIFF because the uncompressed BSDIFF delta is significantly larger than the VCDIFF delta, and reading it from the flash memory takes a lot of time. The larger buffer is at double use here, and since the operation with the read data is extremely simple, the BSDIFF performance gets close to VCDIFF. This is an important factor to consider, since VCDIFF consumes much more memory than BSDIFF, which can be compensated in a larger buffer.

Table 5: Crossbow TelosB specifications.

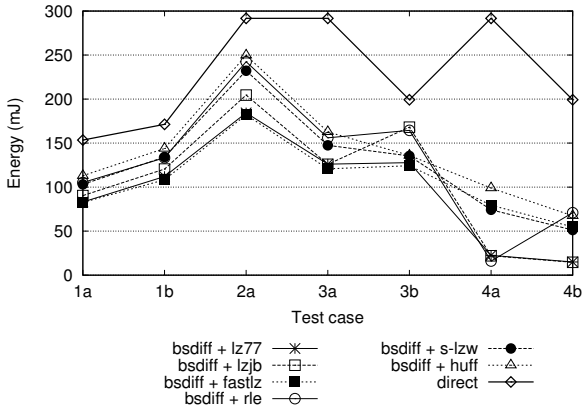
Current drawn - sending (I_{send})	17.4 mA
Current drawn - receiving (I_{recv})	23 mA
Current drawn - processor (I_{proc})	1.8 mA
Input voltage (V)	3 V

4.6 Energy estimation

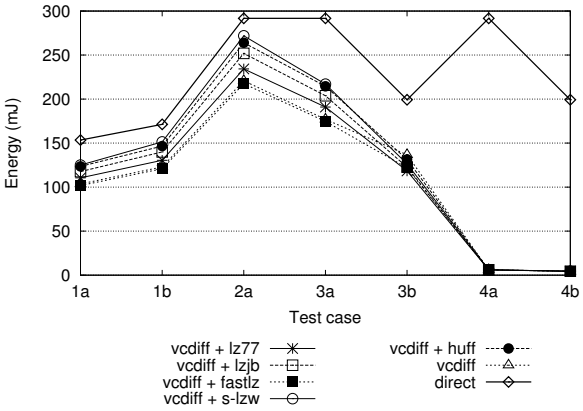
By replacing the specific values for the Crossbow TelosB sensor nodes (table 5) in the energy model described in 4.1, the energy for sending, receiving and processing equals:

$$E_{send} = 0.365mJ$$
$$E_{recv} = 0.483mJ$$
$$E_{cpu} = t_{cpu} * 5.94mJ$$

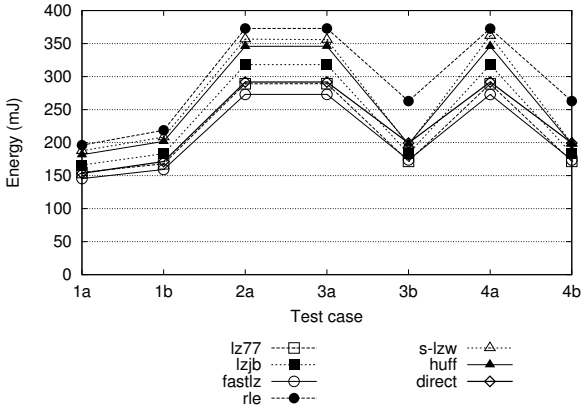
According to these values, processing time of 1 second consumes as much power as sending and receiving 7 data frames. This is the separation point which decides whether additional processing is beneficial to direct transmission.



(a) Decompression + BSDIFF

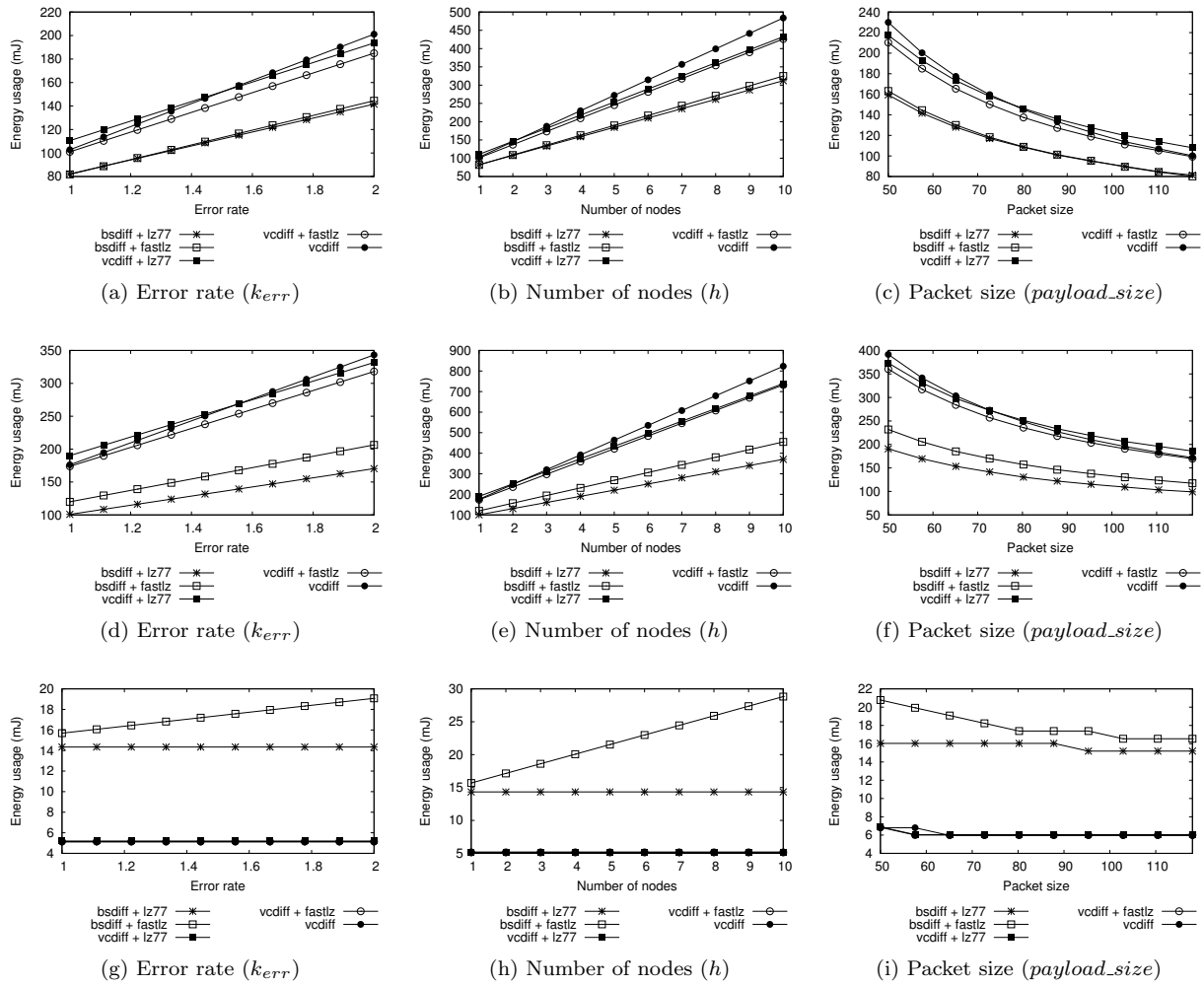


(b) Decompression + VCDIFF



(c) Only decompression

Figure 14: Energy estimation using only decompression (c) and both patching and decompression (a, b). (Constants: $h = 1$, $k_{err} = 1$, *payload_size* $= 114$, *buffer size* $= 128$). "Direct" shows the energy usage of transmitting the data directly, without processing.



(a) Error rate (k_{err}) (b) Number of nodes (h) (c) Packet size ($payload\_size$)

(d) Error rate (k_{err}) (e) Number of nodes (h) (f) Packet size ($payload\_size$)

(g) Error rate (k_{err}) (h) Number of nodes (h) (i) Packet size ($payload\_size$)

Figure 15: Influence of error rate (k_{err}), number of nodes (h) and packet size ($payload\_size$) on energy usage. Computed for test cases 1a (a-c), 3a (d-f) and 4a (g-i), using a 128 byte buffer.

The energy usage for updating one node ($h = 1$), in an ideal environment ($k_{err} = 1$) with 114 byte packets ($payload\_size$) and 128 byte buffer, is shown in figure 14.

For reprogramming one node, using only compressed updates (figure 14c) is more energy efficient than sending data directly, only when FastLZ is used. The additional processing introduced by LZ77 in some cases pays off for the savings in data for transmission; all other algorithms require more energy.

On the other hand, the combination of any compression algorithm with either BSDIFF (figure 14a) or VCDIFF (figure 14b) results in significant reductions in energy usage. For test cases 1a to 3b, highest energy savings are achieved using BSDIFF in combination with LZ77 or FastLZ, while for test cases 4a and 4b, the lowest energy usage is registered using only VCDIFF.

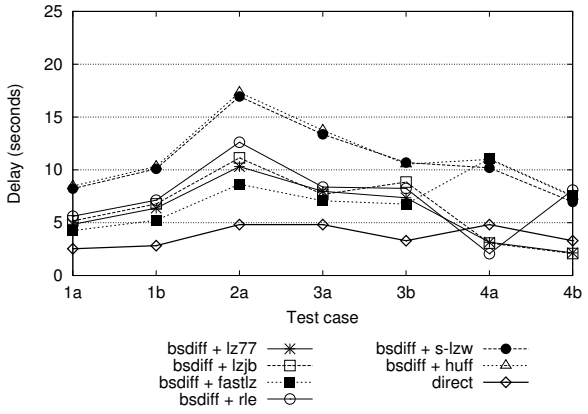
All compression algorithms perform reasonably well when used in combination with BSDIFF. Reductions in energy usage vary between 14% in the worst case, and up to 95% in the best case. The penalty for low compression ratio or extremely high processing time is particularly evident for Run Length and Huffman Encoding, since they provides less energy savings than any of the other algorithms.

VCDIFF has good performance even without using an additional compressor. In fact, only FastLZ reduced the energy usage in all test cases, closely followed by LZ77. In the parameter change test cases, since the VCDIFF delta already fits in one data frame, there is no need to apply additional compression to it.

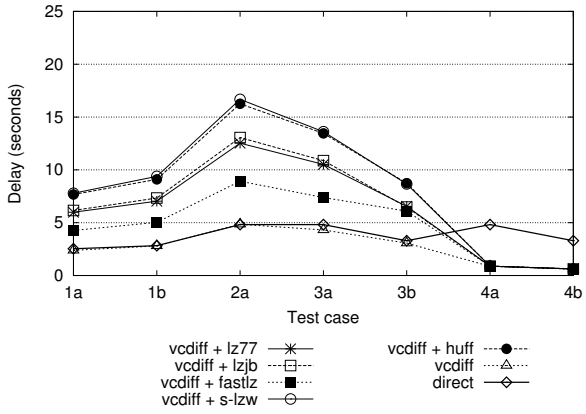
The number of neighbours in a network, retransmission rate of the wireless link and payload size used in the model are external factors. They are either pre-set in the network, or defined by the physical

properties of the deployment area. As shown in figure 15, they can make significant difference in the selection of an algorithm.

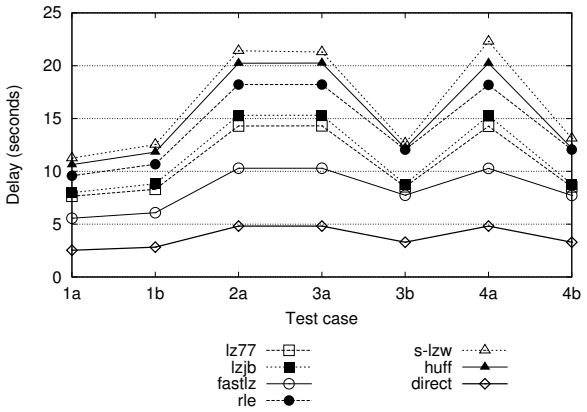
As the number of nodes in the network increases, the channel gets noisier or the packet size decreases, the additional compression is favourable to faster execution time. This is evident in two cases. First, LZ77 has lower energy footprint compared to FastLZ when used in combination with BSDIFF. Second, using only VCDIFF requires more energy than adding additional compression through FastLZ or LZ77.



(a) Decompression + BSDIFF



(b) Decompression + VCDIFF



(c) Only decompression

Figure 16: Delay estimation using only decompression (c) and both patching and decompression (a, b). (Constants: $h = 1$, $k_{err} = 1$, *payload_size* $= 114$, *buffer size* $= 128$). "Direct" shows the delay of transmitting the data directly, without processing.

4.7 Delay

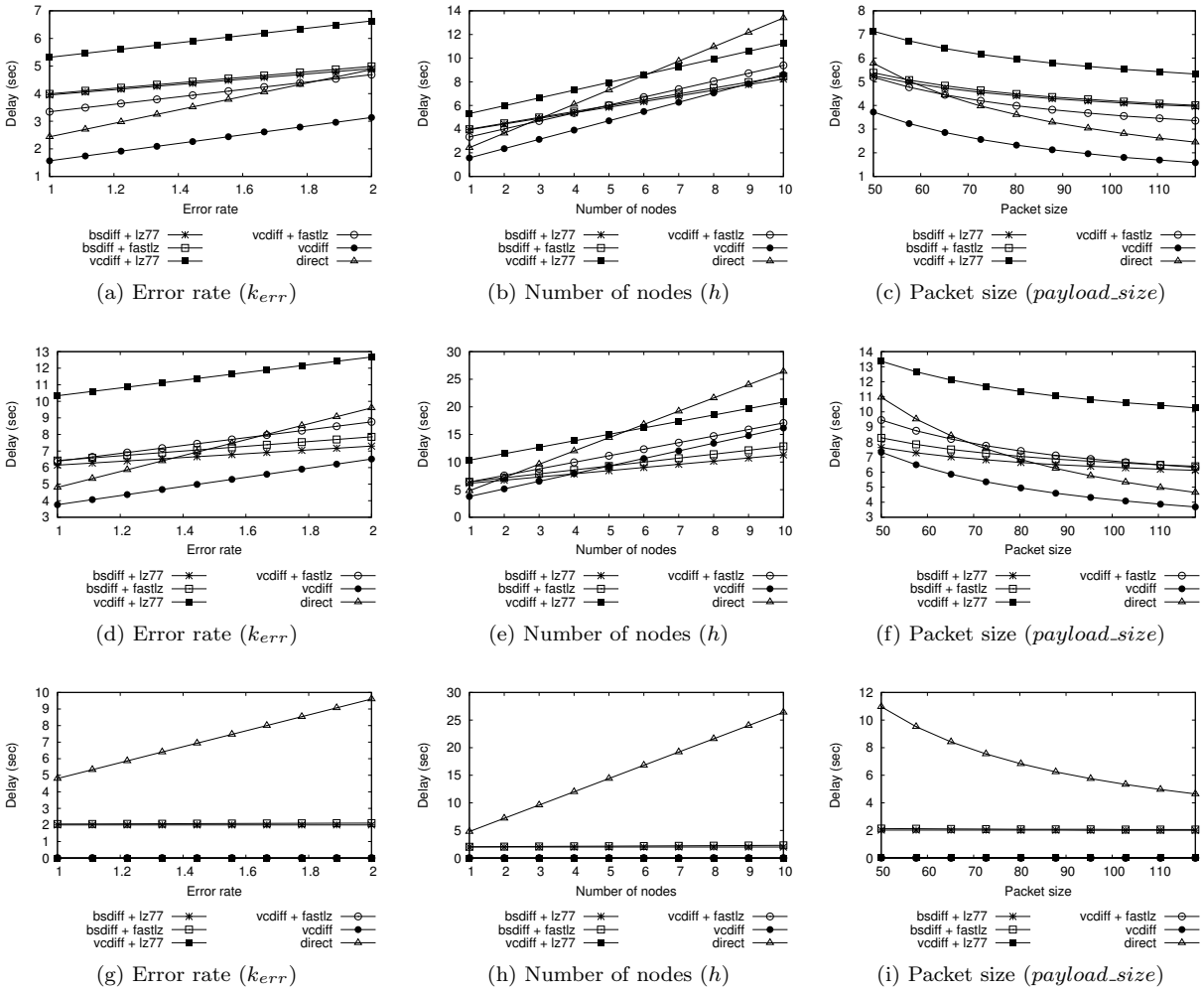
The time needed for updating one Crossbow TelosB node ($h = 1$), in an ideal environment ($k_{err} = 1$) with 114 byte packets (*payload_size*) and 128 byte buffer, is shown in figure 16.

For reprogramming one node, using only compressed updates (figure 16c) is much slower than sending the data directly. Even the fastest algorithm, FastLZ, requires twice as more time.

Slightly improved results are obtained when incremental updates are used - the time required for patching and decompression is larger than the time required to send the difference in data. This is evident both for BSDIFF (figure 16a) and VCDIFF (figure 16b) combinations in test cases 1a to 3b. Using only VCDIFF proves to be the only feasible option in these cases, introducing very little overhead.

In test cases 4a and 4b, the processing overhead is significantly smaller compared to the transmission savings. Therefore, using LZ77, FastLZ or LZJB with BSDIFF, as well as only VCDIFF, is faster than transmitting the entire binary data.

As shown in figure 17, the constants in the delay formula have significant impact. In general, as the number of nodes for update increases, the benefits of additional compression make up for the increased processing time. The tipping point is different for all test cases, but notable changes can be seen starting from two nodes. The impact of payload size and channel retransmission rate is smaller, but still influential.



(a) Error rate (k_{err}) (b) Number of nodes (h) (c) Packet size ($payload\_size$)

(d) Error rate (k_{err}) (e) Number of nodes (h) (f) Packet size ($payload\_size$)

(g) Error rate (k_{err}) (h) Number of nodes (h) (i) Packet size ($payload\_size$)

Figure 17: Influence of error rate (k_{err}), number of nodes (h) and packet size ($payload\_size$) on delay. Computed for test cases 1a (a-c), 3a (d-f) and 4a (g-i), using a 128 byte buffer.

5 Discussion

The results presented in this report suggest that reprogramming of wireless sensor networks can be improved in terms of energy efficiency and time required for update by using data compression and incremental updates. Improvements vary depending on the selection of specific algorithms.

Simply adding compression during reprogramming does not lead to decrease in energy performance or faster updates. In fact, as shown in section 4, some compression algorithms can degrade performance. Compression ratio for all algorithms is lower than reported in other studies, mostly due to the different type of data being compressed. Binary data may not be as compressible as sensed data.

In contrast, using incremental updates showed solid results in all test cases. Up to 95% in energy savings were registered, along with 95% faster updates. Even though highest improvements were registered during parameter reconfiguration, the fact that reduction of 35% in energy consumption was the minimum measured in specific configurations, gives strong arguments for using incremental updates in wireless sensor networks.

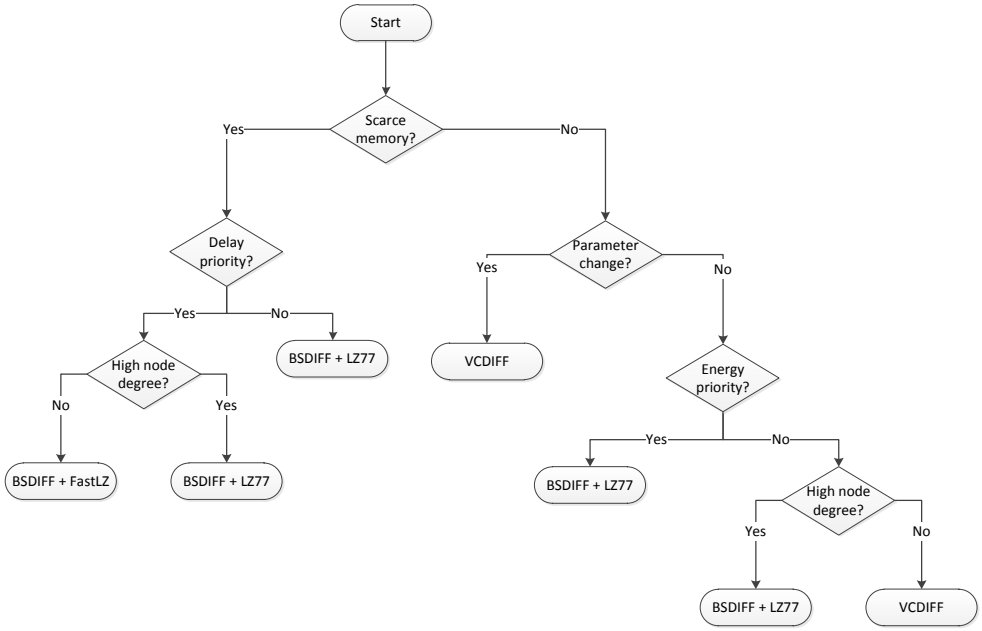


Figure 18: Guideline for selecting the best option for incremental update.

Selecting the best approach for incremental updates depends on the particular implementation. The four factors that play a role in the selection are available resources, type of update, network topology and optimization goal (energy or delay). The three viable solutions are using BSDIFF with either LZ77 or FastLZ, or using only VCDIFF. The entire decision tree is shown in figure 18.

The memory footprint of VCDIFF is a lot higher than BSDIFF with either LZ77 or FastLZ. Therefore, if resources are scarce, VCDIFF is the least acceptable solution.

On the other hand, VCDIFF has incomparable performance in terms of both energy usage and delay when simple changes like parameter change are considered. If most updates are of this type, then VCDIFF is the option to use.

In case updates are more heterogeneous, the number of nodes in the network is small, and delay is a priority, then VCDIFF is again the best option. When energy usage is a priority, or the network is fairly large, then BSDIFF with LZ77 gives the best performance. Finally, when memory is scarce, the network is small and delay is a priority, then BSDIFF with FastLZ is the preferred option.

The energy and delay estimation model presented in this report can be made more precise. At the time being, it does not include MAC protocol behaviour, which can heavily influence the outcome of both metrics. Furthermore, 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. This timing may be crucial to the function of the wireless network itself, and proper isolation should be taken into account. In such cases CPU intensive algorithms might be considered inappropriate.

Decompression and patching time can be decreased by extending the buffer to cover reading instructions from the serial flash. This is a per-algorithm dependency which would result in a larger code base, but would provide better performance.

Significant improvements can be reached by optimizing the incremental update process. One approach would be to adapt the original data, i.e. firmware or executable binary, in such way that it becomes as constant as possible between different versions. This can be achieved through function call indirection [19], extracting and ordering of global variables etc. As a result, the delta scripts would be much smaller, hence easier to compress.

Another approach would be segment an entire update into a set of independent patches. A complete

upgrade to a new version would consist of applying all patches in a specific order. If two different binary versions of the same application exists, intended for different nodes within the network, they can share a subset of the patches, which will be distributed to all nodes within the network. This way the network would be able to handle code differences much easier.

Finally, compression is only the initial step in energy-efficient reprogramming of wireless sensor networks. A lot of open issues remain to be solved, such as integrating compression in existing or new protocols for distributing images and application updates in wireless sensor networks. An additional challenge is to complete all of the previously mentioned tasks in the limited memory that is available in wireless sensor nodes, with as little code overhead as possible. Making this task feasible and integratable will enable easier large-scale, long term deployment of wireless sensor networks.

6 Conclusion

In this report we investigated two approaches for efficient reprogramming of wireless sensor networks. Firstly, we evaluated the performance of general purpose data compression algorithms applied directly on binary data. Secondly, we compared three algorithms for incremental update using delta scripts and combined them with the previously analyzed compression algorithms. Further tests were done on wireless sensor nodes, measuring memory requirements, code footprint, execution time, energy usage and delay.

Results show that data compression in combination with incremental update can significantly decrease energy usage and delay in reprogramming wireless sensor networks. The best option to perform incremental updates depends on multiple factors, for which we have provided a decision tree. Best performance was measured when using either the VCDIFF delta encoding algorithms, or the combination of BSDIFF for delta encoding and LZ77 or FastLZ for decompression.

Acknowledgement

The authors would like to thank Martijn van den Heuvel and Richard Verhoeven for their valuable discussions and improvements on this article, and to Nicolas Tsiftes for kindly providing the source code to the VCDIFF decoder.

This work is supported in part by the Dutch P08 SenSafety Project, as part of the COMMIT program.

References

[1] K. Sha, W. Shi, and O. Watkins, "Using wireless sensor networks for fire rescue applications: Requirements and challenges," in *Electro/information Technology, 2006 IEEE International Conference on*, pp. 239 –244, may 2006.

[2] E. Cayirci and T. Coplu, "Sendrom: sensor networks for disaster relief operations management," *Wirel. Netw.*, vol. 13, pp. 409–423, June 2007.

[3] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson, "Wireless sensor networks for habitat monitoring," in *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, WSNA '02, (New York, NY, USA), pp. 88–97, ACM, 2002.

[4] D. Musiani, K. Lin, and T. S. Rosing, "Active sensing platform for wireless structural health monitoring," in *Proceedings of the 6th international conference on Information processing in sensor networks*, IPSN '07, (New York, NY, USA), pp. 390–399, ACM, 2007.

[5] S. Coleri, S. Y. Cheung, and P. Varaiya, "Sensor networks for monitoring traffic," in *In Allerton Conference on Communication, Control and Computing*, 2004.

[6] A. Dunkels, B. Grnvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *Proc. IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, 11 2004.

[7] J. Polastre, R. Szewczyk, and D. Culler, "Telos: enabling ultra-low power wireless research," in *Proc. Int. Symp. on Information processing in sensor networks*, IPSN, IEEE Press, 2005.

[8] Q. Cao, T. Abdelzaher, J. Stankovic, and T. He, "The liteos operating system: Towards unix-like abstractions for wireless sensor networks," in *Proc. of the 7th int. conf. on Information processing in sensor networks*, IPSN '08, (Washington, DC, USA), pp. 233–244, IEEE Computer Society, 2008.

[9] H. Cha, S. Choi, I. Jung, H. Kim, H. Shin, J. Yoo, and C. Yoon, "Retos: resilient, expandable, and threaded operating system for wireless sensor networks," in *Proc. of the 6th int. conf. on Information processing in sensor networks*, IPSN '07, (New York, NY, USA), pp. 148–157, ACM, 2007.

[10] P. Levis, S. Madden, J. Polastre, R. Szewczyk, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, "Tinyos: An operating system for sensor networks," in *in Ambient Intelligence*, Springer Verlag, 2004.

[11] I. Crossbow Technology, "Mote in-network programming user reference version 20030315," 2003.

[12] 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. Symp. on Networked Systems Design and Implementation - Volume 1*, pp. 2–2, USENIX, 2004.

[13] J. W. Hui and 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*, pp. 81–94, ACM, 2004.

[14] T. Stathopoulos, J. Heidemann, and D. Estrin, "A remote code update mechanism for wireless sensor networks," tech. rep., Center for Embedded Networked Sensing, 2003.

[15] S. Kulkarni and L. Wang, "Mnp: Multihop network reprogramming service for sensor networks," in *Distributed Computing Systems, 2005. ICDCS 2005. Proceedings. 25th IEEE int. conf. on*, pp. 7 –16, june 2005.

[16] R. Panta, I. Khalil, and S. Bagchi, "Stream: Low overhead wireless reprogramming for sensor networks," in *Proc. Int. Conf. on Computer Communications, INFOCOM*, pp. 928 –936, may 2007.

[17] C. Miller and C. Poellabauer, "Reliable and efficient reprogramming in sensor networks," *ACM Trans. Sen. Netw.*, vol. 7, pp. 6:1–6:32, August 2010.

[18] J. Jeong and D. Culler, "Incremental network programming for wireless sensors," in *Conf. on Sensor and Ad Hoc Communications and Networks, IEEE SECON*, pp. 25 – 33, oct. 2004.

[19] R. K. Panta, S. Bagchi, and S. P. Midkiff, "Efficient incremental code update for sensor networks," *ACM Trans. on Sensor Networks*, vol. 7, pp. 30:1–30:32, February 2011.

[20] N. Reijers and K. Langendoen, "Efficient code distribution in wireless sensor networks," in *Proc. ACM Int. Conf. on Wireless sensor networks and applications*, WSNA '03, (New York, NY, USA), pp. 60–67, ACM, 2003.

[21] K. Klues, C.-J. M. Liang, J. Paek, R. Musăloiu-E, P. Levis, A. Terzis, and R. Govindan, "Tosthreads: thread-safe and non-invasive preemption in tinyos," in *Proc. of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys '09, (New York, NY, USA), pp. 127–140, ACM, 2009.

[22] P. J. Marrón, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, and K. Rothermel, "Flexcup: A flexible and efficient code update mechanism for sensor networks," in *In Proc. of the Third European Workshop on Wireless Sensor Networks (EWSN 2006*, pp. 212–227, 2006.

[23] P. Levis and D. Culler, "Maté: a tiny virtual machine for sensor networks," in *Proc. of the 10th int. conf. on Architectural support for programming languages and operating systems*, ASPLOS-X, (New York, NY, USA), pp. 85–95, ACM, 2002.

[24] R. Bosman, J. Lukkien, and R. Verhoeven, "An integral approach to programming sensor networks," in *6th IEEE Consumer Communications and Networking Conf., CCNC*, pp. 1 –5, jan. 2009.

[25] K. Dolfus and T. Braun, "An evaluation of compression schemes for wireless networks," in *Int. Cong. on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT)*, pp. 1183 –1188, oct. 2010.

[26] N. Tsiftes, A. Dunkels, and T. Voigt, "Efficient sensor network reprogramming through compression of executable modules," in *Conf. on Sensor, Mesh and Ad Hoc Communications and Networks, SECON*, pp. 359 –367, june 2008.

[27] M. Nelson, *The Data Compression Book*. New York, NY, USA: Henry Holt and Co., Inc., 1991.

[28] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. on Information Theory*, vol. 23, pp. 337 – 343, may 1977.

[29] J. Bonwick, "Lzjb compression algorithm," 1998.

[30] A. Hidayat, "Fastlz, free, open-source, portable real-time compression library," 2011.

[31] C. M. Sadler and M. Martonosi, "Data compression algorithms for energy-constrained devices in delay tolerant networks," in *Proc. Int. Conf. on Embedded networked sensor systems*, SenSys, pp. 265–278, 2006.

[32] J. Seward, "A program and library for data compression. bzip2 and libbzip2," 2010.

[33] A. Tridgell, "Efficient algorithms for sorting and synchronization," 2000.

[34] D. Korn, J. MacDonald, J. Mogul, and K. Vo, "The VCDIFF Generic Differencing and Compression Data Format." RFC 3284 (Proposed Standard), June 2002.

[35] J. Macdonald, "Xdelta - open-source binary diff," 2011.

[36] C. Percival, "Naive differences of executable code," 2003.

[37] D. Albu, J. Lukkien, and R. Verhoeven, "Energy effect of on-node processing of ecg signals," in *Consumer Electronics (ICCE), 2010 Digest of Technical Papers Int. Conf.*, pp. 7 –8, jan. 2010.

[38] T. R. Burchfield, S. Venkatesan, and D. Weiner, "Maximizing throughput in zigbee wireless networks through analysis, simulations and implementations," 2007.

[39] Jennic, "Calculating 802.15.4 data rates." Application note JN-AN-1035, 08 2006.

[40] N. Tsiftes, A. Dunkels, Z. He, and T. Voigt, "Enabling Large-Scale Storage in Sensor Networks with the Coffee File System," in *Proceedings of the 8th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN 2009)*, (San Francisco, USA), Apr. 2009.

A Detailed information on input data

Algorithm	Binary		bsdiff		rdiff		vcdiff	
	size	c_ratio	size	c_ratio	size	c_ratio	size	c_ratio
huff	17,563	14.84	9,318	54.82	17,573	14.79	11,535	44.07
fastlz	16,653	19.25	8,486	58.85	16,664	19.20	11,390	44.77
lz77	16,201	21.45	8,048	60.98	16,212	21.39	11,172	45.83
lzjb	17,642	14.46	8,871	56.99	17,656	14.39	12,007	41.78
rle	20,736	0.00	10,679	48.22	20,746	0.00	13,345	35.29
s-lzw	17,911	13.15	8,030	61.06	17,917	13.13	11,636	43.58
bzip2	12,917	37.37	6,619	67.91	12,924	37.34	9,311	54.85
Delta size	-		21,278		20,635		13,268	
Entropy	6,717		3.313		6.656		6.746	

Table 6: Compression ratio when upgrading from Contiki 2.3 to 2.4 (Test 1a). The starting image is 22.924 bytes long, while the final one is 20.624 bytes long.

Algorithm	Binary		bsdiff		rdiff		vcdiff	
	size	c_ratio	size	c_ratio	size	c_ratio	size	c_ratio
huff	19,585	14.77	12,264	46.63	19,595	14.73	13,733	40.24
fastlz	18,279	20.46	11,413	50.34	18,290	20.41	13,475	41.36
lz77	17,638	23.25	10,973	52.25	17,649	23.20	13,148	42.79
lzjb	19,487	15.20	11,904	48.20	19,499	15.15	14,363	37.50
rle	23,088	0.00	13,527	41.14	23,098	0.00	15,905	30.79
s-lzw	19,907	13.37	10,986	52.19	19,939	13.23	14,190	38.25
bzip2	14,047	38.87	9,061	60.57	14,046	38.88	10,928	52.45
Delta size	-		24,270		22,991		15,813	
Entropy	6.656		3.911		6.676		6.762	

Table 7: Compression ratio when upgrading from Contiki 2.4 to 2.5 (Test 1b). The starting image is 20.624 bytes long, while the final one is 22.980 bytes long.

Algorithm	Binary		bsdiff		rdiff		vcdiff	
	size	c_ratio	size	c_ratio	size	c_ratio	size	c_ratio
huff	33,620	14.04	21,837	44.17	33,634	14.01	24,918	36.29
fastlz	31,553	19.33	19,484	50.18	31,577	19.27	24,422	37.56
lz77	30,406	22.26	18,317	53.17	30,425	22.21	23,811	39.12
lzjb	33,798	13.59	20,563	47.43	33,821	13.53	25,908	33.76
rle	39,415	0.00	24,928	36.27	39,428	0.00	28,779	26.42
s-lzw	34,181	12.61	19,575	49.95	34,235	12.47	25,666	34.38
bzip2	24,276	37.93	14,736	62.32	24,288	37.90	19,622	49.83
Delta size	-		39,748		39,126		28,503	
Entropy	6.780		4.316		6.780		6.878	

Table 8: Compression ratio when installing a new application by flashing the firmware (Test 2a). The starting image is 22,980 bytes long, while the final one is 39.112 bytes long.

Algorithm	Binary	
	size	c_ratio
huff	18,648	30.19
fastlz	19,037	28.73
lz77	17,918	32.92
lzjb	19,471	27.11
rle	28,485	0.00
s-lzw	17,933	32.87
bzip2	13,417	49.77
File entropy	5.431	

Table 9: Compression ratio when installing an new application (Test 2b). The partial executable is 26,712 bytes long.

Algorithm	Binary		bsdiff		rdiff		vcdiff	
	size	c_ratio	size	c_ratio	size	c_ratio	size	c_ratio
huff	33,620	14.04	12,030	69.24	33,634	14.01	20,001	48.86
fastlz	31,553	19.33	11,652	70.21	31,577	19.27	19,470	50.22
lz77	30,406	22.26	9,375	76.03	30,425	22.21	19,116	51.12
lzjb	33,798	13.59	11,947	69.45	33,821	13.53	20,672	47.15
rle	39,415	0.00	15,823	59.54	39,428	0.00	22,739	41.86
s-lzw	34,181	12.61	10,034	74.35	34,235	12.47	20,230	48.28
bzip2	24,276	37.93	7,406	81.06	24,288	37.90	16,251	58.45
Delta size	-		39,580		39,126		22,550	
Entropy	6.780		2.116		6.780		6.954	

Table 10: Compression ratio when updating an application by flashing the firmware (Test 3a). The starting image is 37,796 bytes long, while the final one is 39,112 bytes long.

Algorithm	Binary		bsdiff		rdiff		vcdiff	
	size	c_ratio	size	c_ratio	size	c_ratio	size	c_ratio
huff	18,648	30.19	10,911	59.15	18,658	30.15	11,804	55.81
fastlz	19,037	28.73	12,438	53.44	19,049	28.69	12,669	52.57
lz77	17,918	32.92	11,990	55.11	17,933	32.87	11,851	55.63
lzjb	19,471	27.11	12,903	51.70	19,483	27.06	12,496	53.22
rle	28,485	0.00	17,106	35.96	28,495	0.00	19,318	27.68
s-lzw	17,933	32.87	10,638	60.18	18,023	32.53	11,097	58.46
bzip2	13,417	49.77	8,919	66.61	13,456	49.63	9,210	65.52
Delta size	-		27,396		26,723		17,589	
Entropy	5.431		2.959		5.432		5.176	

Table 11: Compression ratio when updating an application using partial executables (Test 3b). The starting partial executable is 25,784 bytes long, while the final one is 26,712 bytes long.

Algorithm	Binary		bsdiff		rdiff		vcdiff	
	size	c_ratio	size	c_ratio	size	c_ratio	size	c_ratio
huff	33,620	14.04	4,905	87.46	1,893	95.16	78	99.80
fastlz	31,552	19.33	479	98.78	1,861	95.24	55	99.86
lz77	30,406	22.26	88	99.78	1,853	95.26	55	99.86
lzjb	33,800	13.58	1,336	96.58	1,993	94.90	60	99.85
rle	39,415	0.00	481	98.77	2,069	94.71	62	99.84
s-lzw	34,181	12.61	2,024	94.83	1,903	95.13	64	99.84
bzip2	24,262	37.97	75	99.81	1,758	95.51	112	99.71
Delta size	-		39,130		2,061		60	
Entropy	6.780		0.004		6.410		4.8670	

Table 12: Compression ratio when modifying a parameter in an application by flashing the firmware (Test 4a). Both the starting and final images are 39,112 bytes long. Modification is in two bytes.

Algorithm	Binary		bsdiff		rdiff		vcdiff	
	size	c_ratio	size	c_ratio	size	c_ratio	size	c_ratio
huff	18,648	30.19	3,353	87.45	1,700	93.64	79	99.70
fastlz	19,035	28.74	331	98.76	1,625	93.92	54	99.80
lz77	17,918	32.92	78	99.71	1,657	93.80	54	99.80
lzjb	19,473	27.10	908	96.60	1,806	93.24	58	99.78
rle	28,485	0.00	333	98.75	2,136	92.00	60	99.78
s-lzw	17,933	32.87	1,385	94.82	1,700	93.64	62	99.77
bzip2	13,421	49.76	72	99.73	1,507	94.36	107	99.60
Delta size	-		26,724		2,061		58	
Entropy	5.431		0.006		5.777		4.901	

Table 13: Compression ratio when modifying a parameter in an application using partial executables (Test 4b). Both the starting and final partial executable are 26,712 bytes long. Modification is in two bytes.

Science Reports **Department of Mathematics and Computer Science**
Technische Universiteit Eindhoven

If you want to receive reports, send an email to: wsinsan@tue.nl (we cannot guarantee the availability of the requested reports).

In this series appeared (from 2009):

09/01	Wil M.P. van der Aalst, Kees M. van Hee, Peter Massuthe, Natalia Sidorova and Jan Martijn van der Werf	Compositional Service Trees
09/02	P.J.l. Cuijpers, F.A.J. Koenders, M.G.P. Pustjens, B.A.G. Senders, P.J.A. van Tilburg, P. Verduin	Queue merge: a Binary Operator for Modeling Queueing Behavior
09/03	Maarten G. Meulen, Frank P.M. Stappers and Tim A.C. Willemse	Breadth-Bounded Model Checking
09/04	Muhammad Atif and MohammadReza Mousavi	Formal Specification and Analysis of Accelerated Heartbeat Protocols
09/05	Michael Franssen	Placeholder Calculus for First-Order logic
09/06	Daniel Trivellato, Fred Spiessens, Nicola Zannone and Sandro Etalle	POLIPO: Policies & OntoLogies for the Interoperability, Portability, and autOnomy
09/07	Marco Zapletal, Wil M.P. van der Aalst, Nick Russell, Philipp Liegl and Hannes Werthner	Pattern-based Analysis of Windows Workflow
09/08	Mike Holenderski, Reinder J. Bril and Johan J. Lukkien	Swift mode changes in memory constrained real-time systems
09/09	Dragan Bošnački, Aad Mathijssen and Yaroslav S. Usenko	Behavioural analysis of an I²C Linux Driver
09/10	Ugur Keskin	In-Vehicle Communication Networks: A Literature Survey
09/11	Bas Ploeger	Analysis of ACS using mCRL2
09/12	Wolfgang Boehmer, Christoph Brandt and Jan Friso Groote	Evaluation of a Business Continuity Plan using Process Algebra and Modal Logic
09/13	Luca Aceto, Anna Ingolfsdottir, MohammadReza Mousavi and Michel A. Reniers	A Rule Format for Unit Elements
09/14	Maja Pešić, Dragan Bošnački and Wil M.P. van der Aalst	Enacting Declarative Languages using LTL: Avoiding Errors and Improving Performance
09/15	MohammadReza Mousavi and Emil Sekerinski, Editors	Proceedings of Formal Methods 2009 Doctoral Symposium
09/16	Muhammad Atif	Formal Analysis of Consensus Protocols in Asynchronous Distributed Systems
09/17	Jeroen Keiren and Tim A.C. Willemse	Bisimulation Minimisations for Boolean Equation Systems
09/18	Kees van Hee, Jan Hidders, Geert-Jan Houben, Jan Paredaens, Philippe Thiran	On-the-fly Auditing of Business Processes
10/01	Ammar Osaiweran, Marcel Boosten, MohammadReza Mousavi	Analytical Software Design: Introduction and Industrial Experience Report
10/02	F.E.J. Kruseman Aretz	Design and correctness proof of an emulation of the floating-point operations of the Electrologica X8. A case study

10/03	Luca Aceto, Matteo Cimini, Anna Ingolfsdottir, MohammadReza Mousavi and Michel A. Reniers	On Rule Formats for Zero and Unit Elements
10/04	Hamid Reza Asaadi, Ramtin Khosravi, MohammadReza Mousavi, Neda Noroozi	Towards Model-Based Testing of Electronic Funds Transfer Systems
10/05	Reinder J. Bril, Uğur Keskin, Moris Behnam, Thomas Nolte	Schedulability analysis of synchronization protocols based on overrun without payback for hierarchical scheduling frameworks revisited
10/06	Zvezdan Protić	Locally unique labeling of model elements for state-based model differences
10/07	C.G.U. Okwudire and R.J. Bril	Converting existing analysis to the EDP resource model
10/08	Muhammed Atif, Sjoerd Cranen, MohammadReza Mousavi	Reconstruction and verification of group membership protocols
10/09	Sjoerd Cranen, Jan Friso Groote, Michel Reniers	A linear translation from LTL to the first-order modal μ-calculus
10/10	Mike Holenderski, Wim Cools Reinder J. Bril, Johan J. Lukkien	Extending an Open-source Real-time Operating System with Hierarchical Scheduling
10/11	Eric van Wyk and Steffen Zschaler	1st Doctoral Symposium of the International Conference on Software Language Engineering (SLE)
10/12	Pre-Proceedings	3rd International Software Language Engineering Conference
10/13	Faisal Kamiran, Toon Calders and Mykola Pechenizkiy	Discrimination Aware Decision Tree Learning
10/14	J.F. Groote, T.W.D.M. Kouters and A.A.H. Osaiweran	Specification Guidelines to avoid the State Space Explosion Problem
10/15	Daniel Trivellato, Nicola Zannone and Sandro Etalle	GEM: a Distributed Goal Evaluation Algorithm for Trust Management
10/16	L. Aceto, M. Cimini, A.Ingolfsdottir, M.R. Mousavi and M. A. Reniers	Rule Formats for Distributivity
10/17	L. Aceto, A. Birgisson, A. Ingolfsdottir, and M.R. Mousavi	Decompositional Reasoning about the History of Parallel Processes
10/18	P.D. Mosses, M.R. Mousavi and M.A. Reniers	Robustness os Behavioral Equivalence on Open Terms
10/19	Harsh Beohar and Pieter Cuijpers	Desynchronisability of (partial) closed loop systems
11/01	Kees M. van Hee, Natalia Sidorova and Jan Martijn van der Werf	Refinement of Synchronizable Places with Multi-workflow Nets - Weak termination preserved!
11/02	M.F. van Amstel, M.G.J. van den Brand and L.J.P. Engelen	Using a DSL and Fine-grained Model Transformations to Explore the boundaries of Model Verification
11/03	H.R. Mahrooghi and M.R. Mousavi	Reconciling Operational and Epistemic Approaches to the Formal Analysis of Crypto-Based Security Protocols
11/04	J.F. Groote, A.A.H. Osaiweran and J.H. Wesselius	Benefits of Applying Formal Methods to Industrial Control Software
11/05	Jan Friso Groote and Jan Lanik	Semantics, bisimulation and congruence results for a general stochastic process operator
11/06	P.J.L. Cuijpers	Moore-Smith theory for Uniform Spaces through Asymptotic Equivalence
11/07	F.P.M. Stappers, M.A. Reniers and S. Weber	Transforming SOS Specifications to Linear Processes
11/08	Debjyoti Bera, Kees M. van Hee, Michiel van Osch and Jan Martijn van der Werf	A Component Framework where Port Compatibility Implies Weak Termination
11/09	Tseesuren Batsuuri, Reinder J. Bril and Johan Lukkien	Model, analysis, and improvements for inter-vehicle communication using one-hop periodic broadcasting based on the 802.11p protocol

11/10	Neda Noroozi, Ramtin Khosravi, MohammadReza Mousavi and Tim A.C. Willemse	Synchronizing Asynchronous Conformance Testing
11/11	Jeroen J.A. Keiren and Michel A. Reniers	Type checking mCRL2
11/12	Muhammad Atif, MohammadReza Mousavi and Ammar Osaiweran	Formal Verification of Unreliable Failure Detectors in Partially Synchronous Systems
11/13	J.F. Groote, A.A.H. Osaiweran and J.H. Wesselius	Experience report on developing the Front-end Client unit under the control of formal methods
11/14	J.F. Groote, A.A.H. Osaiweran and J.H. Wesselius	Ananlyzing a Controller of a Power Distribution Unit Using Formal Methods
11/15	John Businge, Alexander Serebrenik and Mark van den Brand	Eclipse API Usage: The Good and The Bad
11/16	J.F. Groote, A.A.H. Osaiweran, M.T.W. Schuts and J.H. Wesselius	Investigating the Effects of Designing Control Software using Push and Poll Strategies
11/17	M.F. van Amstel, A. Serebrenik And M.G.J. van den Brand	Visualizing Traceability in Model Transformation Compositions
11/18	F.P.M. Stappers, M.A. Reniers, J.F. Groote and S. Weber	Dogfooding the Structural Operational Semantics of mCRL2
12/01	S. Cranen	Model checking the FlexRay startup phase
12/02	U. Khadim and P.J.L. Cuijpers	Appendix C / G of the paper: Repairing Time-Determinism in the Process Algebra for Hybrid Systems ACP
12/03	M.M.H.P. van den Heuvel, P.J.L. Cuijpers, J.J. Lukkien and N.W. Fisher	Revised budget allocations for fixed-priority-scheduled periodic resources
12/04	Ammar Osaiweran, Tom Fransen, Jan Friso Groote and Bart van Rijnsoever	Experience Report on Designing and Developing Control Components using Formal Methods
12/05	Sjoerd Cranen, Jeroen J.A. Keiren and Tim A.C. Willemse	A cure for stuttering parity games
12/06	A.P. van der Meer	CIF MSOS type system
12/07	Dirk Fahland and Robert Prüfer	Data and Abstraction for Scenario-Based Modeling with Petri Nets
12/08	Luc Engelen and Anton Wijs	Checking Property Preservation of Refining Transformations for Model-Driven Development
12/09	M.M.H.P. van den Heuvel, M. Behnam, R.J. Bril, J.J. Lukkien and T. Nolte	Opaque analysis for resource-sharing components in hierarchical real-time systems - extended version –
12/10	Milosh Stolikj, Pieter J. L. Cuijpers and Johan J. Lukkien	Efficient reprogramming of sensor networks using incremental updates and data compression
12/11	John Businge, Alexander Serebrenik and Mark van den Brand	Survival of Eclipse Third-party Plug-ins