The Impact of Diverse Execution Strategies on Incremental Code Updates for Wireless Sensor Networks

Kai Lehniger and Stefan Weidling
IHP - Leibniz-Institut für innovative Mikroelektronik, Im Technologiepark 25, 15236 Frankfurt (Oder), Germany
{*lehniger, weidling*}*@ihp-microelectronics.com*

Keywords: Wireless Sensor Networks, WSN, Code Update, Over-the-air, Incremental Reprogramming, Delta, Page-based, Low-power.

Abstract: Wireless Sensor Networks (WSNs) may require code updates for a variety of reasons, such as fixing bugs, closing security holes or extending functionality. WSNs typically have limited resources available and wireless updates are costly in terms of energy and can lead to early battery failure. The idea of incremental code updates is to conserve energy by reusing the existing code image on the node and disseminating only a delta file that is generated by differencing algorithms, which can be used to reconstruct the new image. Beyond these differencing algorithms, there are other strategies to minimize the delta, e.g., reconstructing only the changed parts of the image. This paper points out possible implications of diverse execution strategies and gives suggestions. In addition to the usual delta size, the impact on the flash memory was considered. The presented results can be used to select a fitting strategy for a given use case.

1 INTRODUCTION

WSNs consist of many low-cost, low-power devices with harsh resource restrictions, such as limited energy in the form of a battery or limited processing power. Despite these limitations, nodes in field operation must provide update functionality for adding features or fixing bugs.

Incremental code updates attempt to consume as little energy as possible by calculating the differences between two program images and compiling them as instructions summarized as delta file. This file is then disseminated across the network and executed by the corresponding nodes.

The research field of incremental reprogramming strategies is well studied, with approaches such as R3diff (Dong et al., 2013) to generate delta files. Common to all publications is that they try to minimize the size of the delta file, because over-the-air dissemination of the delta file is considered the main drain of energy. This assumption was possible because the execution of the delta file was equivalent for the different approaches. After the image has been received, it has been completely constructed either on an external memory or in a separate part of the internal memory and then copied to its destination.

Meanwhile, alternative delta execution techniques have been introduced in recent publications, for example, by Kachman and Balaz (2016b). These approaches try to eliminate the overhead that occurs when an image is completely reconstructed before being copied in place, especially the parts that have not changed.

Beyond the differencing algorithms, Lehniger et al. (2018) have proposed an approach that has the same goal of minimizing the delta but taking a different path. Namely, a good order is searched for the pages in flash memory in which they are to be updated.

A major consequence of these diverse execution strategies is a different treatment of the internal flash memory, which affects its lifetime and energy consumption. Kachman and Balaz (2016a) have already considered other metrics for comparison.

This paper tries to support the understanding of the state of the art in this research area by pointing out possible implications of diverse execution strategies and giving suggestions. In addition to the usual delta size, the impact on the flash memory is examined. It also studies the impact of the page update order on the diverse execution strategies.

The rest of this paper is structured as follows. Section II describes various incremental code update strategies and categorizes them. Three strategies are described in detail as case studies in Section III. Section IV discusses the possible implications of the

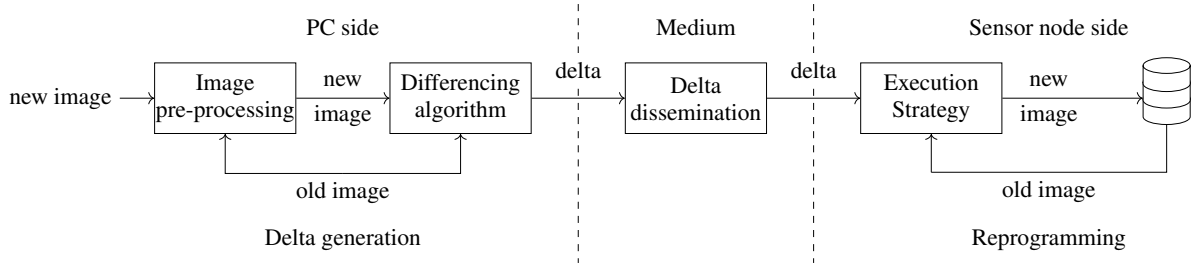

Figure 1: General reprogramming scheme.

strategies with respect to the lifetime of the nodes. The trade-off between dissemination costs and execution costs is evaluated in Section V and Section VI concludes this paper.

2 STATE OF THE ART

The general idea of an incremental code update is to reuse the image already installed on the node. In normal updates/patches, large parts of the code base remain unchanged compared to the previous program version. This applies to the application itself, but more to other parts of the binary file, such as the operating system, drivers, or libraries.

Figure 1 shows the main steps during the code update progress, namely delta generation, dissemination, and reprogramming. Within the first step, preprocessing the image is optional. The idea behind the optimization techniques during the preprocessing is to increase the similarity between the two program images before the actual delta is generated. This can be done in several ways. Koshy and Pandey (2005) add slop regions between functions to compensate function growth. Zephyr (Panta et al., 2009, 2011) and Hermes (Panta and Bagchi, 2009) use indirection tables to reduce the changes that occur to pointers due to function relocation. Hermes uses the indirection table to replace the pointers to the indirection table with the actual function address to minimize the runtime overhead.

The actual delta generation is done by a differencing algorithm. This algorithm and its execution strategy on the node are closely related. Therefore, they are described together. The target of the first step is a delta file. This file generally consists of copy and add instructions. Copies are used to move code to new positions, while add operations add code blocks that were not present in the old binary file. However, the algorithms themselves can be categorized according to their execution strategy.

The first category of algorithms relies on an out-of-place execution strategy. They try to find a sequence of instructions to completely build the new image, either in a separate location in internal memory or in external memory, while leaving the original image untouched. The first representative of this category was based on Rsync (Tridgell, 1999), an algorithm used to exchange binary files over a low-bandwidth link. It slices the binary into blocks and compares its signatures to find unmatched blocks. This resulted in block-based algorithms (Jeong, 2003; Jeong and Culler, 2004) for WSNs, where Jeong and Culler (2004) changed the Rsync algorithm to better handle code shifts. Byte-level algorithms (Brown and Sreenan, 2006; Dong et al., 2011, 2010b,a) make use of the fact that both image versions are known to the host PC. They differ in finding the differences: RMTD (Brown and Sreenan, 2006) uses dynamic programming, DASA (Dong et al., 2011) suffix arrays, R3diff (Dong et al., 2013) footprints, and Dong et al. (2010a) Longest Common Subsequences (LCSs).

The second category of algorithms is characterized by an in-place execution strategy. Instead of reconstructing the images, these algorithms focus on ”fixing” changes. Kachman and Balaz (2016b) point out that for small code changes, many instructions only reconstruct unchanged parts. Their differencing algorithm only updates non-matching segments in the memory. A similar idea, but at the page level, was implemented by Lehniger et al. (2018). Instead of one delta for the complete image, a delta for each page is generated. However, the focus of this approach was not on the differencing algorithm, but how to minimize the overall delta size. This is done by finding a good order for the internal flash memory pages in which they are to be updated.

The remaining part of the general reprogramming scheme is briefly summarized as it is beyond the scope of this paper. During the dissemination step, the delta is split into packets and sent over the air to the sensor nodes. at the receiving node, all packets are stored and checked for completeness and correctness.

3 CASE STUDIES

Before discussing the implications of the two categories of algorithms, it is necessary to better understand the differences between the pursued strategies. For this reason, this section presents three different approaches to an incremental code update as case studies.

The first algorithm, R3diff, was proposed by Dong et al. (2013) and is optimal in terms of the delta size. It compares two binary files at the byte-level. The second algorithm, Delta Generator (DG), proposed by Kachman and Balaz (2016b), is an in-place strategy which reconstructs non-matching segments. The third and last algorithm presented in this section was recently proposed by Lehniger et al. (2018). This algorithm was introduced as an in-place strategy that followed a different approach than Kachman and Balaz.

3.1 R3diff: Optimal Reconstruction

R3diff (Dong et al., 2013) is a classic out-of-place algorithm that achieves a minimal delta. The proposed algorithm calculates an opt_i, a minimum delta size for a given image I for the index i. To get the optimal size of an image, $opt_{|I|}$ is calculated. A method findk is described to find the smallest index k such that $[k, i-1]$ is a segment in the old image. For the calculation of opt_i, opt_i^A and opt_i^C are introduced with $opt_i = min(opt_i^A, opt_i^C)$. opt_i^A describes the minimum delta size if the last instruction was an add, and opt_i^C if it was a copy. The two can be computed as follows:

$$opt_i^A = min(opt_{i-1}^A, opt_{i-1}^C + \alpha + 1)$$

$$opt_i^C = \begin{cases} LARGE\_INTEGER, & if k > i-1 \\ opt_k + \beta, & otherwise \end{cases}$$

With this exhaustive approach and some additional information, it is possible to construct a delta file that is minimal for a given α and β, where α = size of an add instruction (without payload) and β = size of a copy instruction.

As a result, the algorithm has a $O(n^3)$ time and $O(n)$ space complexity, where n is the combined length of the two images.

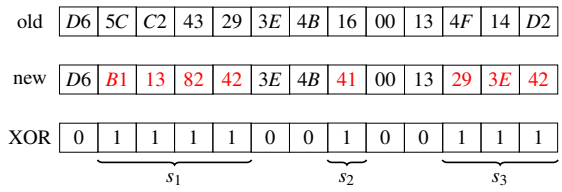

Figure 2: Example for calculating non-matching segments.

3.2 DG: Non-matching Segments

In 2016, Kachman and Balaz (2016b) described their optimized differencing algorithm DG. By reconstructing non-matching segments instead of the full image, they were able to achieve a smaller delta file size. These non-matching segments are determined by a byte-wise XOR of the two images. An example is given in Figure 2. Each block of consecutive 1's, in the result string of the XOR, is a non-matching segment. For each segment, the instructions are generated separately. LCSs are used to create copy instructions. The remaining bytes are added with add instructions. copy instructions must be executed beforehand to prevent their source data from being overwritten by add instructions. For this, the instructions are rearranged. The paper also describes optimization techniques to further reduce the size of the delta.

However, it does not generate an optimal delta. Looking at the byte strings "axb" and "cxd", a diff would find two non-matching segments, and therefore two add instructions would be generated. Each of the add instructions has a size of 6 bytes for a 16-bit architecture. On the other hand, a single add instruction that adds all three bytes has only the size of 8 bytes and saves 4 bytes. Since the optimization techniques focus only on already generated instructions and do not take into account unchanged parts of the image, this optimization cannot be found.

More important than the differencing algorithm itself was the different execution strategy required to perform the update. Unfortunately, that was not part of the work. No description has been given for updating the non-matching segments. The fact that flash memory in general can not be written randomly was completely ignored.

3.3 Page Updates

To write a single byte (or sequence of bytes) to flash memory requires several steps. First, a copy of the block in main memory is needed. Then the operation can be executed on this copy. After that, the block in the flash memory must be erased and the modified copy can then be written to the flash. This would put a tremendous burden on the flash memory itself as each instruction execution of an in-place strategy forces a read-write cycle of a flash memory block. Of course, the copy of the block may be cached in main memory until another memory block needs to be written. In the best case, each block must be written only once.

The algorithm of Lehniger et al. (2018) addresses this problem. Instead of updating segments, it updates a complete page. Pages are described as the smallest

Table 1: Overview of possible implications of different code update executions strategies.

	better ⟵ ⟶ **worse**		
Instruction size	R3diff *next position is given implicitly*	Page-based *extra header, but smaller values*	DG *destination for each instruction*
Number of instructions	DG *only non-matching segments*	Page-based *only pages*	R3diff *always complete image*
Delta size	R3diff *finds optimum for complete image*	DG *only delta for non-matching segments, but no optimizations between non-matching segments*	Page-based *page update can override useful data, depends highly on choosen differencing algorithm*
flash memory usage	Page-based *each block is written once*	R3diff *each block is written once, but external flash is used for image construction*	DG *multiple block writes may be necessary*
Node recovery possibilities	R3diff *old image is valid until new image has been built*		DG, Page-based *in-place execution might corrupt image*

block of memory that can be erased from the flash memory.

If a page has been changed in the new image version, a delta for this page is created. The concrete delta algorithm is unspecified. The main problem with this algorithm is that replacing a page in place could overwrite much of the data that would otherwise have been used for copying. Therefore, a heuristic has been proposed to find an update order for the pages minimizing this overwritten data.

4 THE IMPLICATIONS OF THE DIFFERENT EXECUTION STRATEGIES

In this section, various implications of the different execution strategies are discussed in more detail. Table 1 gives a brief overview. These implications indirectly lead to larger delta sizes, shortened lifetimes, or even node failures.

Encoding Particularities: All featured algorithms use copy and add instructions. In general, the parameters for these instructions are set. Of course, all instructions require some sort of opcode. To copy data, a source address, a destination address, and a length are needed. An add instruction requires a destination address and the payload. In addition, the length of the payload is needed to distinguish the bytes from the next instruction.

Due to the different execution of the instructions, the encoding of the instructions differs. R3diff generates the complete image. In this way, the destination address is implicitly the end of the currently built part of the image. This saves a parameter for each instruction compared to DG. However, because DG fixes only changed parts of the image, the total number of instructions is smaller. It can be stated that for more differences in the two image versions, i.e., more instructions in the delta, the advantage of DG should be smaller.

The other part of the delta is the header. The header contains only information necessary for the execution strategy to correctly interpret the delta and is minimally different between R3diff and DG compared to the size of the remainder of the delta file. However, if a page-based approach is used, each of the page delta files has a header. Having a lot of small changes, i.e., changing many pages with a very small number of instructions, adds a lot of overhead to the headers.

Increased Delta Size Due to Instruction Execution Dependencies: The biggest difference between an in-place and an out-of-place execution strategy lies in the dependency of the instruction execution. While an out-of-place execution leaves the original image untouched until all instructions have been executed, an in-place execution mutates the image with each instruction. This must be taken into account when generating a delta file. This consideration includes, for example, executing copy instructions before adding instructions as described in subsection 3.2. However, this may not be sufficient as copy instructions themselves may depend on each other.

Copy instructions work with two memory areas: the source area, the part of memory from which the data is copied, and the destination area, the part of the memory where the data is copied to. If the destination area of an instruction i overlaps the source area

of an instruction j, the correct execution depends on instruction i, that is, instruction i's execution must not be scheduled prior to j's execution because it would corrupt j's source area.

While this issue is not covered by Kachman and Balaz (2016b), it can be easily solved by modeling these dependencies in a directed graph, where each copy instruction is represented as a node and edges represent dependencies. Topological sorting of the nodes is a valid schedule for the copy instructions. However, cyclic dependencies must be resolved separately, i.e., solving the feedback vertex set problem for the graph that is NP-hard (Karp, 1975).

Another approach might be to associate the instructions execution order and the generated instructions with the certainty of the changes that the other instructions have already made at the source. This had to be done by the algorithm described in subsection 3.3, because the dependencies are even bigger. Instead of source and destination areas of single instructions, complete pages are updated, and each copy instruction that is used to update a page has source areas. These dependencies are too complex to handle with the previous approach because of too many conflicts. By shifting the dependency problem into page order with the described heuristic by Lehniger et al. (2018) and generating instructions thereafter, no instruction reordering is necessary; not even copy instructions need to be executed before add instructions.

Increased Execution Times and Power Consumption due to Excessive Flash Memory Access: While the approach of Kachman and Balaz (2016b) ensures that only changed parts of the binary are encoded, it is still necessary to erase and rewrite every changed block of memory in the flash memory. On the other hand, if an image is constructed in the external flash memory and written to the internal flash when the delta is completely decoded, mass erasure for the full flash memory is possible (if supported).

Using a mass-erase or deleting individual memory blocks can affect execution time and power consumption. However, the various execution strategies may otherwise affect the lifetime of the node.

Shortened Flash Memory Lifetime Due to Excessive Write Access: While flash memory reads are harmless to the flash memory, write operations slowly decrease its lifetime. This can be more harmful than energy consumption, as nodes with empty batteries can be collected and reused, while damaged flash memory is much harder to replace or compensate for.

An execution strategy that allows for smaller delta files and saves energy, even if the various memory erase mechanics are taken into account, can still result in a shorter lifetime of the node if it heavily consumes the flash.

While Lehniger et al. (2018) limiting the erase and write cycles to 1 per memory block, the algorithm of Kachman and Balaz (2016b) may require several cycles for individual blocks.

Of course, there are techniques to minimize the burden on flash memory. For example, it is possible to reorder the delta instructions to favor consecutive operations on the same block. In this way, it is possible to perform multiple operations before the memory block is written to the flash. However, the reordering is either limited by the dependencies between the operations or has a negative affect on the size of the delta file.

Hardware-level and software-level techniques, such as caching and paging (Hennessy and Patterson, 2011; Panda et al., 2001; Mittal, 2014; Gracioli et al., 2015), for minimizing memory access are well researched. Paging algorithms have been developed to manage the pages of main memory. If there are no free pages left and a process needs more memory, a page must be swapped out and written to the hard drive. This can be adapted to the problem of the delta execution. Multiple memory blocks can be cached in main memory, and if the cache is full and a new block needs to be modified, a block is written to the flash. The big advantage of the delta execution compared to memory access of processes is that the order of operations is known. Therefore, an optimal strategy can be found.

However, the main memory in microcontrollers is small and blocks can consume large parts of it. It may be impossible to have a cache with more than one handful of blocks, which limits this method.

Node Recovery Possibilities: Recovery mechanisms during the update process are necessary for safety and security reasons. During this process, there are several points for checks to ensure its correctness. Whenever a check fails, a recovery mechanism becomes active, either to repeat the step hoping to complete the update, or to return the node to its pre-update state.

These mechanisms often include hash calculations and re-transmissions of certain parts of the delta. For this purpose, the node must be able to perform these actions, i.e., the image must still be intact. After parts of the image have been overwritten, it can no longer be assumed that routines will function properly until the update is complete.

While overwriting the image is the final step during the execution of an out-of-place update, it will

start earlier during an in-place update. For example, there is no way[1] to verify the correctness of the decoded image in an in-place update without overwriting parts of the old image, because the execution of the first instruction already causes alteration. An attacker could use this vulnerability to manipulate an update so that it does not detect the manipulation before it can not be recovered. Therefore, security aspects should be considered before deciding on an execution strategy.

5 TEST METHODOLOGY

When comparing different delta generation algorithms, it is common to use a real application and apply changes of different sizes to it. This can start with changing a constant and can end with replacing the entire application with another. The test cases can then be ordered by their impact on the image, i.e., the number of bytes that have changed.

However, the number of changes in the code does not directly match the changes in the image. Modern compilers can perform many complex optimization techniques that can marginalize changes. For example, encapsulating some functionality in a function may still result in the exact same binary if the compiler decides to inline this function.

On the other hand, small changes that affect the size of a function or even a single base block can lead to large code shifts and pointer alterations that go along with it.

These inconsistencies due to the indirection called compiler make it very difficult to create test cases that consistently scale in one or more desired metrics. This could be the already mentioned byte change, but also metrics like the image size or the number of shifts.

To control all these metrics, this paper uses images generated in addition to images based on real applications. The images are generated as a random byte sequence based on different hardware platforms that differ in flash size and even in address space size. Based on an generated or compiled image, code shifts and mutations of different sizes can be applied.

Of course, all semantics and dependencies between the bytes for generated images and changes are lost. Instead of a shifted function only a few bytes are shifted. Instead of a changing constant or a pointer, only randomly selected bytes are altered. A shift does not mean that pointers are altered elsewhere, as is the case with a function shift. However, the algorithms studied do not consider any of this information. It is still possible to get examples that are impossible with real world applications, such as large code shifts without alterations. However, it is still interesting to see how the algorithms react to those cases and scale to find advantages and disadvantages.

Another advantage is the ability to create many similar image pairs for testing to eliminate the possibility that any random structure will undesirably favor an algorithm.

6 RESULTS

For the tests, we used randomly generated images ranging in size from 1kB up to 64kB (up to 128kB for time measurements). We have made changes to

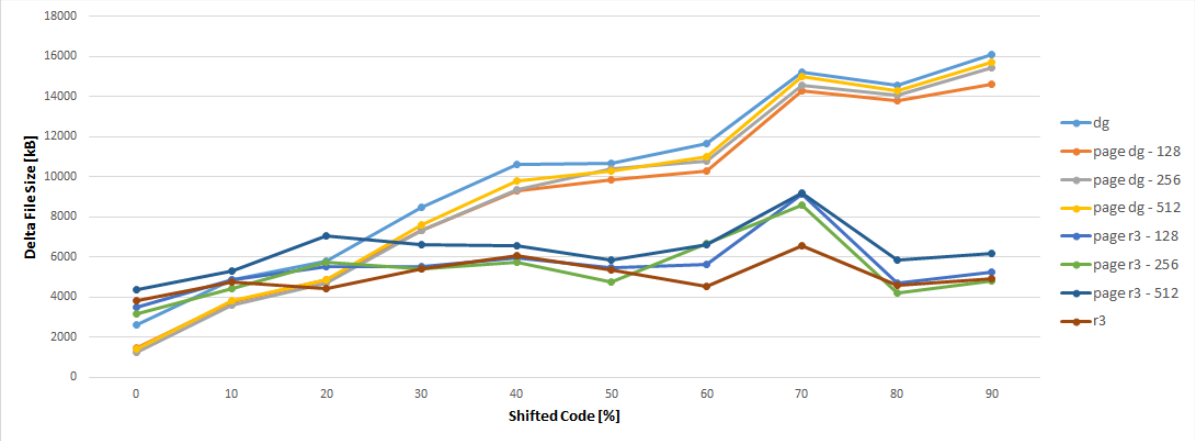

Figure 3: Delta file sizes in bytes for different percentages of shifted code and 2% mutations.

[1] Of course, it would be possible to create a copy of the image and perform the update on the copy. However, this is essentially a transformation to an out-of-place update.

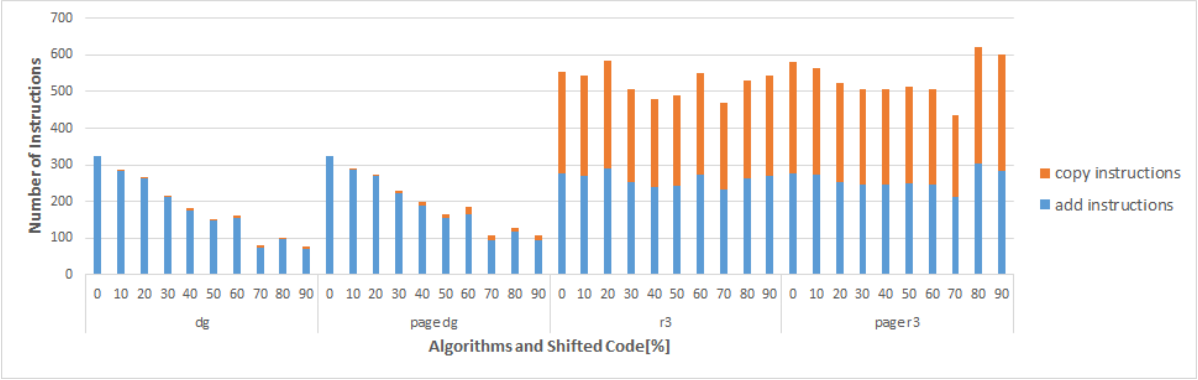

Figure 4: Distribution of copy and add instructions for different algorithm combinations and code shifts.

these images as described in the previous section. The applied changes refer to the size of the image, beginning with no code shifts, up to 90% code shifts. For random mutations, 2% were used because code shifts already introduce new code when the remaining areas are filled. For charts, data from the 16kB images was used. However, the results for other image sizes are similar.

The algorithms of section 3 have been implemented. Since the page-based approach does not define a concrete differencing algorithm, both R3diff and DG were used. In addition, three different page sizes are examined: 128 bytes (e.g., ATmega32U4 microcontroller), 256 bytes, and 512 bytes (e.g., MSP430F5438A microcontroller).

Figure 3 shows the different delta sizes in bytes in perspective to an increasing code shift rate. First, it can be stated that the differencing algorithms itself performs relatively the same, regardless of whether the page-based approach has been integrated or not, or how large the page size is. Second, while DG performs very well when small changes occur, the delta files become significantly larger as more data is shifted compared to R3diff.

Although the effect of page size, as mentioned earlier, is low, there are differences. Especially DG can be improved because of the better encoding. For R3diff, the results for the different page sizes vary more, but 256 bytes give the best results. This is because it is possible to encode offsets and lengths with only one byte[2]. While this also applies to 128-byte pages, the header overhead is larger, resulting in worse delta files. For larger microcontroller address spaces, three or four bytes are needed to encode an offset or a length. Then the benefits will be even greater. Also DG is more affected because its instruction has an additional parameter that benefits from the better encoding.

The reason why DG worsens with larger deltas can be seen in Figure 4. This figure shows the number of add and copy instructions used in the delta file. In general, DG uses far fewer instructions, as this was the main intent for the development of this algorithm. More noteworthy is the decreasing number of add instructions with increasing code shifts for DG and the overall low number of copy instructions. Combined

[2]Lengths of 256 can be encoded with 0 because a copy of 0 bytes is meaningless.

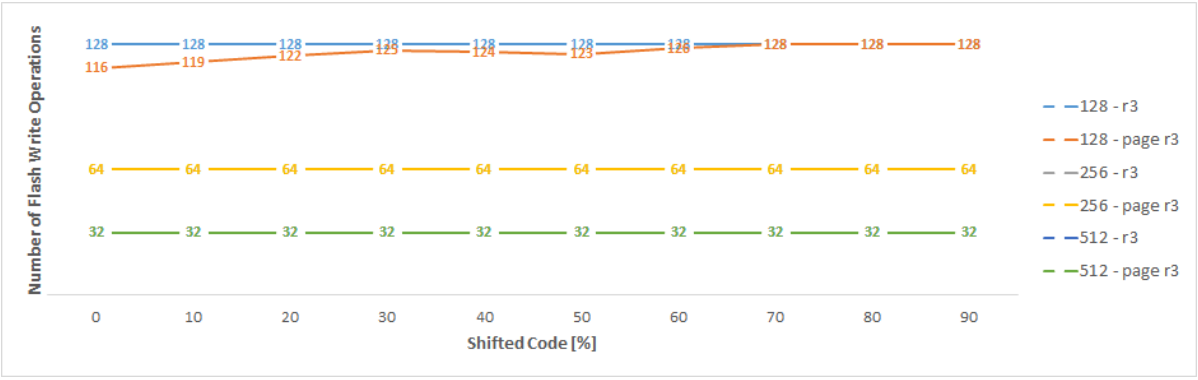

Figure 5: Page writes fro R3diff with and without page approach.

Figure 6: Page writes for DG with and without page approach.

with the fact that the DG delta file gets very large, it can be concluded that DG is unable to optimize for many small changes. Instead, it only generates a bunch of some-byte add instructions for each non-matching segment. With increasing code shifts, these non-matching segments grow together and the total number of add instructions decreases, and the algorithm can also use a few copies. However, the number of bytes added with one add instruction increases dramatically. In the end, almost the entire image is rebuilt with add instructions.

Looking at page write operations for the internal flash, there are almost no differences for R3diff (see Figure 5). With R3diff, each page is written exactly once when the reconstructed image is flashed. In its paged counterpart, only the affected pages are written. Due to the page size and the changes in the images due to the 2% mutations, all pages are affected for page sizes of 256 and 512. Only for the small 128 byte pages some pages remain unchanged.

For DG, an instruction execution caching mechanism has been implemented. One page can be kept in main memory. If successive instructions write data to the same page, this will only be done in main memory. Only when the cache needs to be flushed, a write operation is performed in flash memory. If no cache is used, each instruction would result in at least one page write.

Figure 6 shows the results for these cached operations. Some pages need to be written more than once because of dependencies, although the impact of these dependencies is far less than expected. In any case, more page writes are required by DG than by R3diff. It should be noted that the 2% mutations

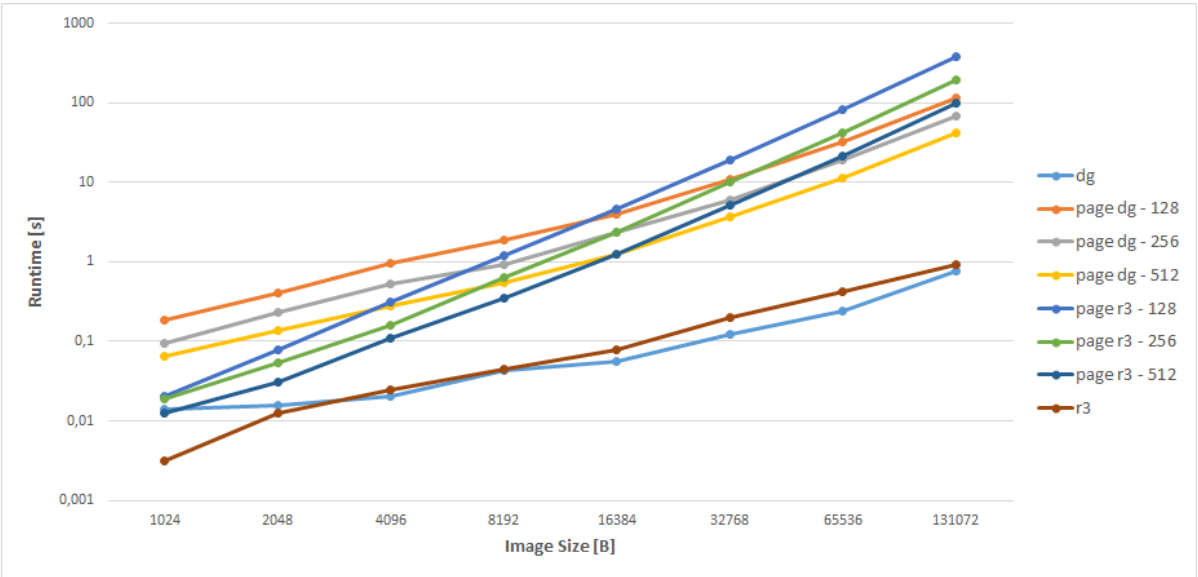

Figure 7: Algorithm runtimes for different image sizes in seconds.

that are randomly distributed are not suitable for DG because they affect almost all pages and negate the benefits that DG has. With increasing code shifts, dependencies on the instructions cause the instructions to reorder, increasing page writes.

Despite all the advantages and disadvantages of the algorithms, time can be a limiting factor. Figure 7 shows the average runtime of all algorithms for different image sizes. Both scales are logarithmic. While the basic algorithms itself scale with a similar complexity ($O(n^3)$ for non-paged and $O(n^4)$ for paged algorithms), the page algorithms are one to three orders of magnitude slower. This is because the algorithms have to be executed for each page, but the input size is only halved. Larger images have more pages, which increases the distance during runtime. A positive fact is that the delta calculation for each page can be done independently after the page order has been calculated leaving much potential for parallelization.

7 CONCLUSION

This paper studied the implications of diverse execution strategies for incremental code updates. These range from larger delta files through shortened lifetimes to node failures. While the original idea of incremental code updates is to conserve energy by propagating only a delta file generated by differencing algorithms, the effects on the sensor node's flash memory are also relevant.

To study the impact of these diverse execution strategies on incremental code updates, the flash memory accesses were included in the consideration, in addition to the usual delta size.

First, it could be observed that the differencing algorithms themselves behave relatively the same regardless of whether the page-based approach was integrated or not, or how big the page size is. Although the effect of the page size is low, there are differences between the different execution strategies, for example, through better encoding of offsets and lengths. With larger address spaces of microcontrollers, the differences may be even greater.

After all, the runtime of an algorithm is also crucial and can be a limiting factor. Page-based algorithms are one to three orders of magnitude slower, but it should be mentioned that the delta computation for each page could be parallelized after the page update order is determined, which could speed up the runtime.

In conclusion, while a reduced delta size can minimize transmission costs, reducing flash access may also increase the lifetime of sensor nodes by not unnecessarily wearing the flash memory. And the code update execution strategy is critical. In the future, with growing complexity for embedded systems, algorithms must be able to handle larger images. Due to the recent trends for general-purpose processors with stagnating clock rates and increasing number of cores, algorithms that can be parallelized are highly beneficial for those future applications.

ACKNOWLEDGEMENTS

This work was supported by the Federal Ministry of Education and Research (BMBF) under research grant number 03IPT601X.

REFERENCES

Brown, S. and Sreenan, C. (2006). Updating software in wireless sensor networks: A survey. Technical report, Dept. of Computer Science, National Univ. of Ireland, Maynooth.

Dong, W., Chen, C., Liu, X., Bu, J., and Gao, Y. (2011). A lightweight and density-aware reprogramming protocol for wireless sensor networks. *IEEE Transactions on Mobile Computing*, 10(10):1403–1415.

Dong, W., Liu, Y., Chen, C., Bu, J., and Huang, C. (2010a). R2: Incremental reprogramming using relocatable code in networked embedded systems. In *IEEE INFOCOM 2010*. IEEE.

Dong, W., Liu, Y., Wu, X., Gu, L., and Chen, C. (2010b). Elon: enabling efficient and long-term reprogramming for wireless sensor networks. *ACM SIGMETRICS Performance Evaluation Review*, 38(1):49–60.

Dong, W., Mo, B., Huang, C., Liu, Y., and Chen, C. (2013). R3: Optimizing relocatable code for efficient reprogramming in networked embedded systems. In *INFOCOM, 2013 Proceedings IEEE*, pages 315–319. IEEE.

Gracioli, G., Alhammad, A., Mancuso, R., Fröhlich, A. A., and Pellizzoni, R. (2015). A survey on cache management mechanisms for real-time embedded systems. *ACM Computing Surveys (CSUR)*, 48(2):32.

Hennessy, J. L. and Patterson, D. A. (2011). *Computer architecture: a quantitative approach*. Elsevier.

Jeong, J. (2003). Node-level representation and system support for network programming.

Jeong, J. and Culler, D. (2004). Incremental network programming for wireless sensors. In *IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks (SECON)*, pages 25–33.

Kachman, O. and Balaz, M. (2016a). Effective over-the-air reprogramming for low-power devices in cyber-physical systems. In *Doctoral Conference on Computing, Electrical and Industrial Systems*, pages 284–292. Springer.

Kachman, O. and Balaz, M. (2016b). Optimized differencing algorithm for firmware updates of low-power devices. In *19th IEEE International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pages 1–4. IEEE.

Karp, R. M. (1975). On the computational complexity of combinatorial problems. *Networks*, 5(1):45–68.

Koshy, J. and Pandey, R. (2005). Remote incremental linking for energy-efficient reprogramming of sensor networks. In *Proceeedings of the Second European Workshop on Wireless Sensor Networks*, pages 354–365.

Lehniger, K., Weidling, S., and Schölzel, M. (2018). Heuristic for page-based incremental reprogramming of wireless sensor nodes. In *2018 IEEE 21st International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*. IEEE.

Mittal, S. (2014). A survey of techniques for improving energy efficiency in embedded computing systems. *International Journal of Computer Aided Engineering and Technology*, 6(4):440–459.

Panda, P. R., Catthoor, F., Dutt, N. D., Danckaert, K., Brockmeyer, E., Kulkarni, C., Vandercappelle, A., and Kjeldsberg, P. G. (2001). Data and memory optimization techniques for embedded systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 6(2):149–206.

Panta, R. K. and Bagchi, S. (2009). Hermes: Fast and energy efficient incremental code updates for wireless sensor networks. In *IEEE INFOCOM 2009*, pages 639–647. IEEE.

Panta, R. K., Bagchi, S., and Midkiff, S. P. (2009). Zephyr: Efficient incremental reprogramming of sensor nodes using function call indirections and difference computation. In *Proc. of USENIX Annual Technical Conference*.

Panta, R. K., Bagchi, S., and Midkiff, S. P. (2011). Efficient incremental code update for sensor networks. *ACM Trans. Sen. Netw.*, 7(4):30:1–30:32.

Tridgell, A. (1999). *Efficient algorithms for sorting and synchronization*. PhD thesis, Australian National University Canberra.