An Efficient Code Update Solution for Wireless Sensor Network Reprogramming

Biswajit Mazumder, Jason O. Hallstrom
School of Computing
Clemson University
Clemson, SC, USA
{bmazumd, jasonoh}@cs.clemson.edu

ABSTRACT

We present an incremental code update strategy used to efficiently reprogram wireless sensor nodes. We adapt a linear space and quadratic time algorithm (*Hirschberg's* algorithm) for computing *maximal common subsequences* to build an *edit map* specifying an edit sequence, required to transform the code running in a sensor network to a new code image. We then present a heuristic-based optimization strategy for efficient edit script encoding to reduce the edit map size. Finally, we present experimental results to demonstrate the reduction in data size to reprogram a network using this mechanism. The approach achieves reductions of 99.987% for simple changes, and between 86.95% and 94.58% for more complex changes, compared to full image transmissions — leading to significantly lower energy costs for wireless sensor network reprogramming. We compare the results with reductions achieved by other incremental update strategies described in prior work.

Categories and Subject Descriptors

C.3 [**Special-purpose and application-based systems**]: Real-time and embedded systems

Keywords

Incremental code update, reprogramming, longest common subsequence, wireless sensor networks, code distribution.

1. INTRODUCTION

Wireless sensor networks (WSNs) typically consist of resource-contrained sensor nodes to enable low power consumption and longer operation times. The applications are developed and compiled on typical desktop systems, and then programmed to flash program memory of the nodes using an in-system-programmer (ISP), or other serial device reprogramming approach [10, 15]. Both approaches can program only one sensor node at a time, causing network programming time to increase linearly with network size. Many WSNs, e.g. disaster management, structural health monitoring, and volcanic activity monitoring [1, 9, 25], may require a large number of nodes to be deployed, rendering these programming approaches unusable.

The ability to reprogram nodes is also essential in maintaining large WSN deployments. If new functionality must be added, or changes must be made to correct bugs after a large network has been deployed, the in-system-programming and serial reprogramming approaches can be prohibitively time-consuming. Network reprogramming using wireless communication to transfer program images to sensor nodes mitigates the problems posed by these approaches. While wireless methods of data dissemination enable sensor devices freedom from direct connections to the host system, wireless data transmission is energy intensive. Previous studies have reported that transmission of a single bit of data requires 1,000 times the energy required for the execution of a single instruction [20, 21, 26]. Brute force update mechanisms which use wireless communication to reprogram nodes are not energy efficient, as they require the entire code image to be transmitted throughout the network. Incremental code update strategies significantly reduce the amount of data that must be transferred to the boot loader, thus improving the energy footprint of the network. Integrating incremental update strategies can result in faster and more efficient network reprogramming.

We present an incremental code update mechanism which transmits an *edit map* encoding the differences between old and new program images. We are able to generate the differences between the two files using a divide-and-conquer dynamic programming approach. Our incremental code update solution does not use block level code comparison, and hence is able to locate and send differences at byte-level granularity. Our approach is also independent of any program code structure knowledge, and thus provides a platform and programming language independent solution.

We adapt Hirschberg's algorithm [3] (used for computing *maximal common subsequences*) to compute the differences between two program image files. The adapted code differencing algorithm is capable of generating the *diff*s between the two files in $O(n^2)$ time and $O(n)$ space, where n is the length of the new program image. The first step of the incremental code update strategy, i.e. the code differencing algorithm, is run on a standard desktop system to avoid computationally-expensive operations from executing on the sensor nodes.

In the second step, the differences are encoded in an edit map using heuristic-based optimization. The optimization

.

strategy efficiently encodes the edit map using the least possible number of bytes. The edit map is then propagated to the resource-constrained sensor nodes using a standard data dissemination algorithm. The nodes are responsible for decoding the edit map and performing the required data write, and/or move operations to update the program image. Since edit map creation, propagation, and decoding are decoupled, our strategy can be adapted for use with any data dissemination protocol.

We conduct experiments for various representative code update scenarios and present corresponding results to demonstrate the reduction in reprogramming data and energy footprint we are able to achieve over simple reprogramming strategies which involve transmission of full program images. We also compare the results with reductions achieved by other incremental code update strategies described in prior work. We demonstrate significant data and power savings over the state-of-the-art incremental update strategies under multiple code update scenarios.

2. RELATED WORK

Levis et al. present TinyOS [13], which provides Crossbow Network Programming (XNP) [8, 23] as its network reprogramming implementation. XNP achieves network reprogramming by broadcasting the entire program image to nodes in a single-hop network. Culler et al. present Deluge [4], a reliable data dissemination protocol which also propagates complete binary images. However, both protocols are inefficient; there are often common code segments between versioned images.

Stathopoulos et al. present Multihop Over-the-Air Programming (MOAP) [22], which uses a data dissemination protocol called *Ripple* to distribute code to sensor devices. Unlike network *flooding*, Ripple selectively forwards packets to nodes while utilizing a sliding window protocol for controlling retransmissions. Nodes have the ability to transmit parts of the program code they have already received to new nodes, while waiting for retransmission of lost packets.

Levis et al. present Maté [12], which deals with network reprogramming by transmitting application-specific code for execution on a virtual machine. While this allows Maté to be significantly faster during reprogramming, the approach is not useful when the virtual machine itself needs to be reprogrammed. Levis et al. also present Trickle [14], which uses an epidemic-based data dissemination protocol to avoid flooding the network as in Maté. However, none of these approaches consider incremental code updates for efficient reprogramming.

Jeong describes Fixed Block Comparison (FCB) [6], which divides program images into fixed size blocks and compares the blocks in the corresponding locations in both the old and new program images. FCB then propagates only the blocks of code from the new program image which are different from the previous version. FCB performs only marginally better than XNP when the two program image versions are not aligned with each other.

Jeong et al. also present an incremental code update strategy in [7], where they adapt the Rsync algorithm [24] to compute differences between program image versions. The approach again partitions the program image into fixed-size blocks (B bytes), and then uses a checksum pair (checksum, hash value) to represent each block, and stores the pair in a hash table. Next, a sliding window of size B bytes is run on the new program image, and the checksum and hash value for each window are calculated; lookups are performed in the hash table for potential matches. While Rsync is also platform and language independent, there are problems with this approach. The total number of hash computations used in Rsync is proportional to the size of the code image, $O(n)$. Considering that each hash computation requires at least linear time, the time required for all the hash computations and lookups is $O(n^2)$. Second, on a hash match, the approach requires a byte-by-byte scan through the code to avoid false match positives. Finally, the size of the sliding window defines the match granularity in the two image versions. If there are multiple matching image segments of size $(B - b)$ bytes, where $0 < b < B$, this approach would fail to identify the matches.

Panta et al. present Zephyr [17], an incremental reprogramming strategy based on an optimized version of the Rsync algorithm [24], in conjunction with function call indirection. This approach requires application-level code modifications to reduce *function shifts* caused when function bodies are shifted from their original locations between image versions. Next, the optimized Rsync algorithm is used to compute the differences between the two code images, creating a *delta*. While the traditional Rsync algorithm is able to identify matching blocks, the optimized version computes the *maximal super-block* between the two images in $O(n^2)$ time, where n is the length of the code images. A *super-block* comprises contiguous matching blocks, and a maximal super-block is the largest super-block. While Zephyr presents an improvement over the strategy presented in [7], it still shares all the problems of the Rsync-based approach. Further, this solution is not platform and programming language independent; it requires knowledge of program structure.

Munawar et al. present Dynamic TinyOS [16], which uses high-level knowledge of application structure to make application updates. This is achieved using extensions to the NesC compiler which convert TinyOS applications and system components into separate binary objects during compilation. Standard data dissemination protocols are then used to update individual objects. This approach also requires knowledge of program code structure, which reduces its applicability to systems developed using other compilers and languages.

Reijers et al. describe an efficient code distribution strategy that uses a *diff*-like approach to computing the edit script for encoding the differences between two program images [20]. Their approach makes use of a *suffix tree*, which requires $O(n)$ time and space to build, where n is the length of the older version of the program image. However, their approach needs n traversals of the suffix tree for each position of the image vector, thus requiring $O(n^3)$ time. Additionally, the edit script encoding scheme is complex, requiring a large number of commands and opcodes, and is architecture specific.

In contrast to all the prior approaches, our incremental update strategy uses an adapted version of the Hirschberg's algorithm to compute the differences between program images. Hirschberg's algorithm has quadratic time and linear space complexity and employs a divide-and-conquer dynamic programming approach to compute a globally optimal subsequence between two strings. We adapt Hirschberg's algorithm to build the edit map containing the edit script re-

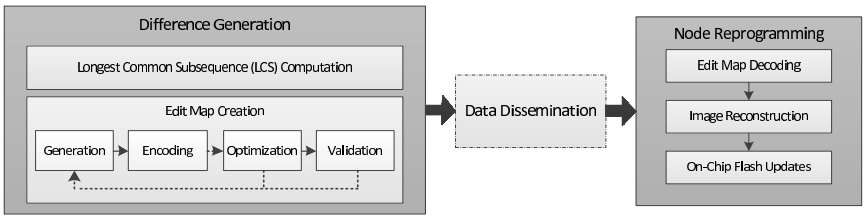

Figure 1: Incremental network reprogramming workflow

quired to transform the code running in the network to a new code image. Since we do not use a block-based approach, our solution is able to identify even small code segments which match between the program images. Further, we present an optimization strategy for encoding the edit map which significantly reduces the amount of data that needs to be transmitted (and the energy expended) for successful sensor node reprogramming.

3. DESIGN AND IMPLEMENTATION

The incremental code update strategy presented in this paper builds upon the idea that an application program image can be thought of as a byte string of length n. When the program image for a sensor node needs to be updated, the *maximal common subsequence*, also known as the *longest common subsequence* (LCS), between the two strings is computed. The substrings of the new program image which are not part of the LCS constitute the image data that must be transmitted to the sensor nodes.

Figure 1 illustrates the workflow for the reprogramming strategy. The *difference generation* phase, which involves computation of the LCS and the associated edit map, is computed at the host system side, as it is the most resource-intensive phase of the process. This requires the host system to have access to the previous version of the program image. In the event the host does not have access to the previous image version, it attempts to retrieve the image from a node within the network, assuming that all nodes run identical images. In the event the host is unable to access a previous version of the image, it proceeds with normal network programming. The *data dissemination* phase is initiated by the host system using a standard data dissemination protocol, such as XNP or Deluge, over a wireless radio link. We do not explore any of the data dissemination algorithms as part of our approach; these are beyond the scope of discussion in this paper. The final *node reprogramming* phase is the responsibility of the sensor network.

3.1 Difference Generation

The difference generation phase consists of two sub-phases. The first involves computing the LCS between the two program image versions. The second sub-phase uses the LCS to prepare and encode the *edit map* containing the edit script to be transmitted to the sensor network.

3.1.1 Longest Common Subsequence.

Consider an arbitrary node in a sensor network. Let the version of the program image currently installed on the sensor node be defined as $X = x_1x_2x_3...x_m$, where $|X|$ is m and x_i is a byte at offset i in the image. Let the new program version be n bytes long, and be defined as $Y = y_1y_2y_3...y_n$. Hirschberg's algorithm finds the string $L = l_1l_2l_3...l_r$, such that L is a common subsequence of X and Y, and its length r ($|L|$) is maximized. Let the set of prefixes of the strings X and Y be $\{X_1, X_2, X_3, ..., X_m\}$ and $\{Y_1, Y_2, Y_3, ..., Y_n\}$, respectively, where X_i and Y_i are the prefixes of size i bytes.

Let $LCS(X_i, Y_j)$ denote the LCS for the prefixes X_i and Y_j. If we denote $|LCS(X_i, Y_j)|$ as $C(i, j)$, then the dynamic programming formulation for $C(i, j)$ is as follows:

$$C(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C(i-1, j-1) + 1 & \text{if } x_i = y_i; \\ max(C(i-1, j), C(i, j-1)) & \text{if } x_i \neq y_j; \end{cases}$$

Figure 2 presents a logical view of the LCS L; the program image strings X and Y are represented as m and n byte memory blocks, respectively. L is represented by the sequence of shaded boxes labeled $l_1, l_2, l_3, ..., l_r$. The arrows illustrate the mapping from the old to the new program image segments in the LCS.

Hirschberg initially presents an algorithm to calculate the length of the LCS of any two strings ($C(i, j)$) using dynamic programming, along with a memoization-based, bottom-up table building scheme, requiring $O(mn)$ time and $O(mn)$ space [3]. Next, a modified version of this algorithm capable of computing the LCS length in $O(min(m, n))$ space is presented. Using the modified algorithm, Hirschberg finally presents a divide-and-conquer algorithm to compute L. We implemented Hirschberg's algorithm and adapted it so that it accepts program images as input.

When the host system does not have access to the previous version of the program image or is not aware of the size of the previous image version, some subtle changes are made to the reprogramming strategy. When the host system receives a new program image of size n bytes to be programmed, it issues a read command to the boot loader executing on some sensor node in the network. The boot loader reads n bytes from application flash memory and returns the data back to

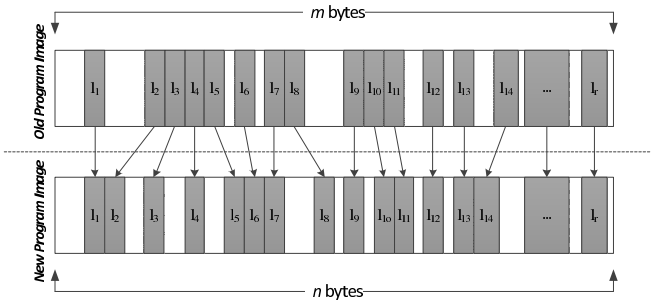

Figure 2: Logical view of LCS between two images

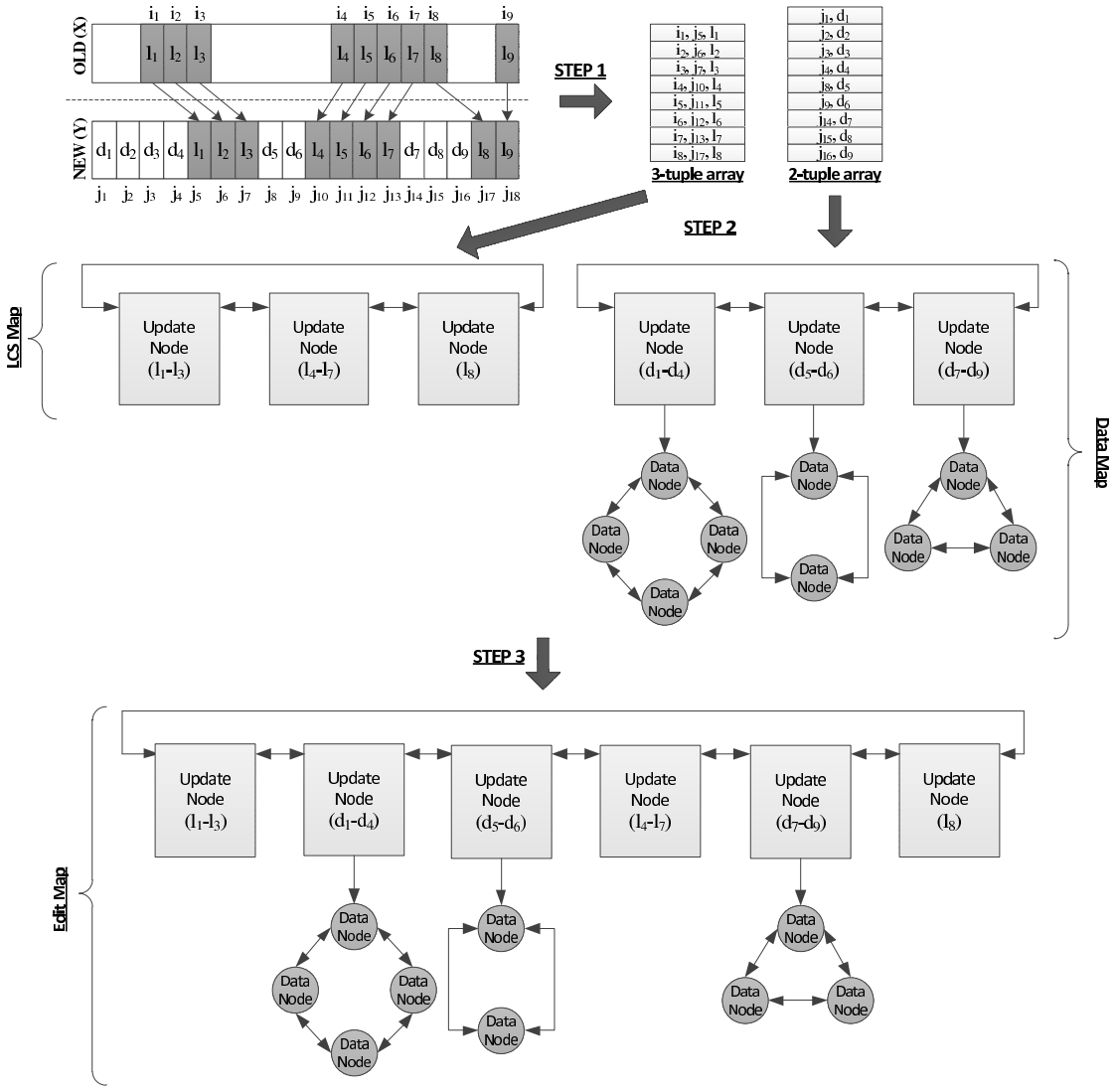

Figure 3: *Edit map* generation flowchart

the host. The host system treats these n bytes of data as the previous version of the program image. Thus, adapted to this scenario, the modified algorithm executes in $O(n^2)$ time and $O(n)$ space.

3.1.2 Edit Map Creation.

The edit map is composed of two different types of data. The first consists of the LCS segments. From a network reprogramming perspective, transferring the LCS segments from the host system to the sensor nodes would be redundant since that data already exists as part of the old program image. However, to reprogram a sensor node, the boot loader needs information about the starting addresses and extents of the LCS segments, along with the destination addresses where the segments need to be relocated. The second type of data consists of the new program image data segments that *do not* belong to the LCS and must be transferred to the sensor node.

The LCS segments can be further classified into two sub-groups. The first sub-group consists of data segments that reside in the same address locations in the old and the new program images; l_1, l_4, l_7, and l_9 in Figure 2, are examples. Since these data segments are already where they need to be, they are not included as part of the edit map. The second sub-group consists of data segments that need to be moved from one address location to another; l_2, l_3, and l_5 are examples.

Map Generation. The output of the adapted algorithm is L, the string containing the longest common subsequence of X and Y. Figure 3 illustrates the steps involved in generating the edit map from the LCS. The first step involves calculating the locations of the LCS segments $l_1, l_2, l_3, ..., l_r$ in X and Y in $O(n)$ time and storing them as an array of 3-tuples, (i, j, l_k), where l_k is the k^{th} element of L, and i and j are its corresponding locations in X and Y, respectively. A 3-tuple entry is not created for cases where $i = j$, e.g. in the case of l_9 in Figure 3. Non-LCS data in Y (the new program image) is stored in an array of 2-tuples, $(j,\ y_j)$, where j is

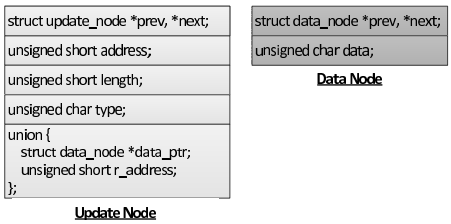

Figure 4: Update and data node structures

the location of the data element and y_j is the j^{th} element. Both arrays are sorted by j.

The second step involves building an *LCS map* and a *data map*; the *update node* and *data node* structures used for creating these maps are shown in Figure 4. Both data structures consist of a doubly linked list of update nodes. Each update node contains address, length, and type fields, as well as a union over a pointer to a data node, and a remote address. Each data node contains one byte of data. The fields are considered in more detail in the remainder of the section.

The LCS map is built by scanning the 3-tuple array and combining LCS segments with contiguous X and Y address locations. Consider LCS segments l_4, l_5, l_6, and l_7 in Figure 3. The starting address locations (i_4, i_5, i_6, and i_7) as well as the destination address locations (j_{10}, j_{11}, j_{12}, and j_{13}) of these four LCS segments are contiguous. Thus, a single update node entry is created for the entire range in the LCS map. The update node contains the starting address of the LCS segment in Y (j_{10}), the starting address of the LCS segment in X (i_4), the length of the segment (4), and a type indicating that the node is being used for the LCS map. Note that the starting address of the LCS segment in X is stored in the `r_address` field within the union (The `data_ptr` field is used in update nodes within the data map). Each LCS map entry corresponds to data that needs to be read from one location and written to a corresponding location.

The data map is similarly created by scanning the 2-tuple array from the previous step. The non-LCS data segments which have contiguous destination address locations (in Y) are identified and merged into individual update nodes, e.g., d_1, d_2, d_3, and d_4, with address locations j_1, j_2, j_3, and j_4 are merged into a single update node. The update node representing such a data segment in the data map contains the starting address of the segment in Y (j_1), the length of the segment (4), and a pointer to a doubly linked list of data nodes. The data nodes store the individual bytes at

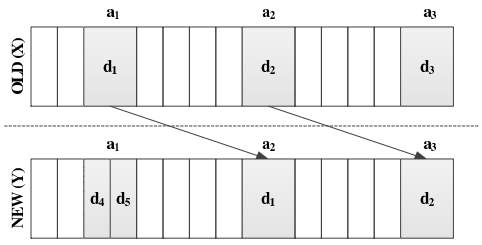

Figure 5: Update ordering problem

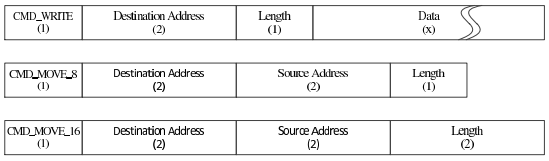

Figure 6: Edit map encoding scheme

each address location (d_1, d_2, d_3, and d_4). Each data map entry corresponds to contiguous data blocks that need to be written in the new program image.

The final step involves merging the data map and the LCS map. The merge process is non-trivial, as it entails a priority ordered merge of the data and the LCS map elements. Incremental reprogramming requires data in flash to be moved in-place using limited RAM. Consider Figure 5, where the top and bottom images depict the state of flash memory before and after reprogramming, i.e. X and Y, respectively. The data segments in X at locations a_1, a_2, and a_3 are d_1, d_2, and d_3, respectively. Data segments d_1 and d_2 are part of the LCS, and hence need to be moved in memory, whereas d_4 and d_5 are new data segments which will be transmitted from the host system.

Assume that data segment d_1 is updated by incoming data segments d_4 and d_5 before it could be moved to location a_2, or that d_1 (in X) is moved to location a_2 (in Y) before d_2 is moved to location a_3. This would result in incorrect reprogramming. To avoid such scenarios, the order in which the data and LCS map elements need to be encoded in the edit map (and then transferred to the sensor nodes) is determined at the host system. The correct priority-based ordering in the scenario shown in Figure 5 is to move d_2, then move d_1, and then finally write d_4 and d_5. The ordering used while merging the data and LCS maps into the edit map prioritizes read operations at address locations in X over write operations at address locations in Y.

Map Encoding. The edit map resulting from the combination of the data and LCS maps is encoded for efficient transmission. The host system uses three instructions to encode the edit map: `CMD_WRITE`, `CMD_MOVE_8`, and `CMD_MOVE_16`, the formats for which are shown in Figure 6. Each of the operations require 1 byte to represent the opcode.

The write operation, `CMD_WRITE`, is used to represent each of the data map elements in the edit map, identified by the type variable in the update node structure in Figure 3. The write operation uses 2 bytes to specify the address location where the write should occur, 1 byte for the length of the data that must be written, and `x` bytes for the data itself, where `x` is the value of the length variable. The maximum amount of data that is transferred in a single message is set to 256 bytes; this allows the use of 1 byte for the length variable (by using the value 0 to represent 256).

The move operations, `CMD_MOVE_8` and `CMD_MOVE_16`, are used to represent each of the LCS map elements in the edit map. They both use 2 bytes each to specify the destination and source address for the LCS segments to be moved. `CMD_MOVE_8` is used to specify move operations for segments with length less than or equal to 256, and hence uses 1 byte for the length variable, while `CMD_MOVE_16` uses 2 bytes for longer segments.

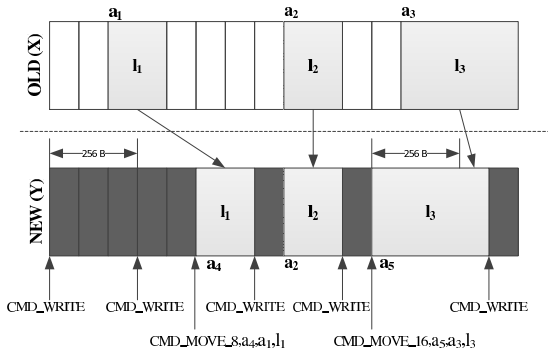

Figure 7: Application of edit operations

Figure 7 illustrates a representative edit scenario. For the LCS segment l_1, a `CMD_MOVE_8` operation is used. A `CMD_MOVE_16` operation is used for the segment l_3, as the length of l_3 is greater than 256. Since LCS segment l_2 is already in its final position, it does not need an update. The remainder of the data segments in the image are populated using `CMD_WRITE` operations. Based on the priority ordering described previously, `CMD_MOVE_8` will be the first operation to be sent, followed by the remaining operations, from left to right, as in the figure.

Map Optimization. The optimization phase begins from the encoding phase and affects the map generation phase in an iterative fashion, as shown in Figure 1. Encoding a single `CMD_MOVE_8` operation requires 6 bytes, regardless of the length of the LCS segment being moved, as shown in Figure 6. Encoding a complete `CMD_WRITE` operation requires $4 + \mathtt{x}$ bytes, where `x` is the length of the data segment being transmitted. A `CMD_MOVE_8` operation with a length value of 2 can be converted into a `CMD_WRITE` with no change in communication cost, while a move operation with a length value of 1 can be converted into a write operation requiring 1 byte less. Converting `CMD_MOVE_8` operations to `CMD_WRITE` operations allows for multiple write operations to be consolidated into one when the segments are contiguous. The optimization step builds on this idea and uses it as a heuristic to reduce the encoded edit map size.

The first step in the optimization phase consists of calculating the cost of the incremental update, i.e. the number of bytes required to encode the edit map, denoted by $\mathtt{C_{update}}$. The update cost is calculated as follows:

$$\mathtt{C_{update}} = 6 * \mathtt{N_{lcs\_map\_8}} + 7 * \mathtt{N_{lcs\_map\_16}} + 4 * \mathtt{N_{data\_map}} + \sum_{i=1}^{\mathtt{N_{data\_map}}} \mathtt{L_i}$$

where $\mathtt{N_{lcs\_map\_8}}$ and $\mathtt{N_{lcs\_map\_16}}$ are the number of LCS map elements in the edit map which use `CMD_MOVE_8` and `CMD_MOVE_16` operations, respectively, $\mathtt{N_{data\_map}}$ is the number of data map elements in the edit map, and $\mathtt{L_i}$ is the length of the data segment contained within the $\mathtt{i^{th}}$ data map element.

In the next step, a copy of the edit map is saved, and the edit map is subjected to a *merge*. Merging involves converting all LCS map entries of size less than or equal to a specified *merge window* to data map entries, and then running a linear scan to consolidate newly formed contiguous data map entries. The value for $\mathtt{C_{update}}$ is recalculated post-merge and compared with its last known value. The merge window value is initially set to 2 for the first merge operation. In subsequent iterations, the edit map is merged, while the window value is incremented by powers of 2, and $\mathtt{C_{update}}$ is recalculated. A binary search is employed to determine the merge window value for which $\mathtt{C_{update}}$ attains a minimum. Once this window has been determined, the saved edit map is subjected to a final merge operation and then encoded for transmission.

Map Validation: The encoded edit map that has been created in the previous phase is checked for errors before it is transmitted to the sensor nodes. Map validation is achieved by decoding the encoded edit map and applying the resulting operations to a copy of the older image version, X. After application of the edit map to X, the resulting image is compared to the desired new image version, Y, using a linear scan in $O(n)$ time. A successful match validates the encoded edit map and initiates the data dissemination phase.

3.2 Data Dissemination

The data dissemination phase involves the generation of fixed-size data packets from the edit map, and their subsequent transmission to sensor nodes using an XNP-like data dissemination protocol implemented in C. The boot loader provides the reprogramming logic and is also reponsible for the reception of these packets. After the packets are received, the edit map is temporarily stored in external memory and node reprogramming is initiated.

3.3 Node Reprogramming

The node reprogramming phase consists of decoding the individual edit map operations and using these operations to reconstruct the new program image. The `CMD_WRITE` and `CMD_MOVE_8` operations are trivial to perform; typically data for a `CMD_WRITE` operation is already available, while the contents of a `CMD_MOVE_8` (up to 256 bytes) can be copied to RAM, and then moved to the new memory location. `CMD_MOVE_16` operations are slightly more complex to perform when the data to move is larger than the RAM capacity, and the starting and ending ranges overlap. Under such circumstances, our approach ensures that updates always occur without destruction of necessary data. An alternate approach is to use an external memory module as a buffer. The data manipulation is done on the external memory, and then moved back to its new location in on-chip flash.

For our implementation of this approach, we use Ferroelectric RAM (FRAM) as the external memory module[1]. FRAMs are characterized by non-volatility, low power consumption (significantly lower than flash memory), faster read and write performance (comparable to SRAM), and a higher number of write-erase cycles [19]. Additionally, FRAMs provide byte addressable memory, like NOR flash devices.

The very first time the sensor node is programmed using an ISP, the program image is also written to the FRAM device. When the edit map is received by the sensor node, the initial data manipulation is done using the FRAM. At the end of the image reconstruction phase, the FRAM contains the updated version of the image, which is then written to on-chip flash memory. This process ensures that when the sensor node is not being reprogrammed, the image on the flash is mirrored on the FRAM.

[1]Using an external memory module saves energy expended on flash reads, since flash writes occur in page granularity.

4. EVALUATION

4.1 Experimental Setup

We implemented the incremental code update reprogramming strategy for the MoteStack [2], a state-of-the-art in-situ sensing platform, which uses an AVR Atmel (ATMega 644P) microcontroller (MCU) operating at 10 MHz, and powered at 3.3V. The MCU consists of 64KB of in-system-programmable flash memory, 2KB of EEPROM, and 4KB of SRAM. The boot loader is installed in the on-chip flash, which offers read-while-write capabilities. We added a 64KB FRAM memory device [19] to use for the image reconstruction phase.

We consider five software change scenarios involving the latest stable version of our custom C-based sensor operating system (with standard OS services) as test cases for our evaluation.

1. **Changing a constant (minor change).** We use a standard blink application as our base case and change a constant to make the LED blink every two seconds (instead of one).

2. **Modification of implementation file (moderate change).** We add 91 lines of (non-whitespace) code to convert the base application into a LED test suite, where various patterns are displayed on five LEDs.

3. **Changing an installed application (major change).** We next write an application to manipulate external flash memory. The new application writes a data buffer filled with random data to external flash memory, and then reads the page back.

4. **Modification of core OS (moderate change).** We next comment out a few lines of code so the new application version does not contain the ZigBee driver module.

5. **Modification of core OS (moderate change).** We next comment out a few lines of code so the new application version does not contain the Wi-Fi driver module.

We also consider five scenarios to evaluate how the approach performs when applied to typical code changes in TinyOS, using standard applications from the `apps` directory of the TinyOS 2.1.0 code distribution[2] so that comparisons can be made between our code update strategy, Zephyr, and Rsync:

6. **Changing a constant (minor change).** We change a constant in the `Blink` application to alter the blinking rate of a LED and reprogram a basic `Blink` application install.

7. **Changing an installed application (major change).** We next replace the installed `Blink` application with the `RadioCountToLeds` application.

8. **Modification of implementation file (moderate change).** We next comment out a few lines of code from `RadioCountToLeds`.

9. **Modification of core OS (moderate change).** We next comment out a few lines of code from `RadioCountToLeds` to remove the `AMControl` module.

10. **Modification of core OS (moderate change).** Finally, we comment out a few lines from `RadioCountToLeds` to remove the `Leds` module.

Data Transmission Savings: We first evaluate the performance of the incremental code update approach. The percentage compression in data size to be transmitted when reprogramming a node (P_{tx}) is calculated as the ratio of the length of the generated edit script ($\mathtt{C_{update}}$) to the length of the new program image version (L_{new}). It is expressed as a percentage, reflecting the percentage of data that is transmitted using the incremental update approach compared to transmitting the full image. The packet overhead during data dissemination is dependent on packet length and protocol; hence it is not considered as part of the evaluation of the edit script generation strategy.

Device Type	Read	Write	Erase
NAND Flash	2.29 nJ/B	14.55 nJ/B	4.03 nJ/B
NOR Flash	2.09 nJ/B	793.75 nJ/B	881.25 nJ/B
FRAM [19]	0.33 nJ/B	0.33 nJ/B	-NA-

Table 1: Energy consumption characteristics

Merge Window Optimization: We next evaluate the effect of varying the merge window size on the size of the generated edit map while using the custom C-based sensor OS. We record the $\mathtt{C_{update}}$ values for fixed merge window sizes of 0 (indicating that a merge will not be conducted), 1, 2, 4, 8, 12, and 16, for cases 1-5. A merge window of w bytes converts LCS map entries of length less than or equal to w into data map segments, so as to consolidate the newly formed contiguous data map entries into a single entry.

Image Reconstruction Cost: Finally, we evaluate the cost of image reconstruction on the sensor node and consider the potential energy savings. Let $\mathtt{h}$ be the length of the program image, $\mathtt{m}$ be the amount of data that needs to be moved in FRAM (due to `CMD_MOVE_8` and `CMD_MOVE_16` operations), and $\mathtt{w}$ be the amount of new data that needs to be written in the FRAM (due to `CMD_WRITE` operations). Let $\mathtt{R_{FRAM}}$ and $\mathtt{W_{FRAM}}$ be the cost of reading and writing a byte of data in FRAM, respectively. Finally, let $\mathtt{W_{FLASH}}$ be the cost of writing a byte in flash, and $\mathtt{C_t}$ is the cost of transmitting a byte wirelessly. The cost of simple reprogramming, $\mathtt{C_r}$, is given by:

$$\mathtt{C_r} = \mathtt{h} * (\mathtt{C_t} + \mathtt{W_{FLASH}})$$

whereas the cost of incremental reprogramming with image reconstruction, $\mathtt{C_i}$, is given by[3]:

$$\mathtt{C_i} = \mathtt{C_{update}} * \mathtt{C_t} + \mathtt{m} * (\mathtt{R_{FRAM}} + \mathtt{W_{FRAM}}) + \mathtt{w} * \mathtt{W_{FRAM}} + \mathtt{h} * \mathtt{W_{FLASH}}$$

Table 1, which is adapted from [11,18,19], presents the energy consumption charactersitics of NAND flash, NOR flash, and FRAM devices for read, write, and erase operations. For wireless data transmission, we consider the XBee low power RF module [5]. Assuming that the XBee module performs at the maximum advertised rate of 250,000 bps, the transmit and receive energy requirements are 4.75 mJ/B and 5.8 mJ/B, respectively. We use the average of the transmit and receive costs, and set $\mathtt{C_t}$ to 5.28 mJ/B.

[2]Code change scenarios 6-10 are replicated from [17].

[3]Cost of executing instructions on the μC is not considered.

	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6	Case 7	Case 8	Case 9	Case 10
L_{old} (B)	37594	37594	38068	38294	37674	2650	2650	11526	11512	11378
L_{new} (B)	37594	38068	38294	37674	35276	2650	11526	11512	11378	11220
$\mathtt{C_{update}}$ (B)	5	4966	9748	2042	2815	6	10241	1121	1194	1173
P_{tx} ($\mathtt{C_{update}} * 100/L_{new}$)	0.013%	13.05%	25.45%	5.42%	7.98%	0.22%	88.5%	9.74%	10.49%	10.46%

Table 2: Edit map sizes for C-based OS and TinyOS upgrade scenarios

Merge Window	Case 1	Case 2	Case 3	Case 4	Case 5
0	5	5033	12436	2097	3173
1	5	5013	12080	2056	3039
2	5	4979	10810	2056	2829
4	5	4966	10036	2042	2825
8	5	5051	9750	2044	2815
12	5	5190	9986	2046	2832
16	5	5424	10471	2052	2888

Table 3: Edit map sizes for different merge windows in cases 1-5

4.2 Results

Data Transmission Savings: Table 2 presents the edit map sizes (C_{update}) and P_{tx} values achieved for both the C-based OS and TinyOS code upgrade scenarios, along with the corresponding old and new image sizes (L_{old} and L_{new}, respectively). In case 1, the difference between the two program images is small, and this is reflected in the small $\mathtt{C_{update}}$ size (5 bytes) and the correspondingly small P_{tx} (0.013%). In case 2, which is a typical change in the software development life cycle of an embedded device, the edit map size was 4966 bytes, resulting in a P_{tx} of 13.05%.

In case 3, which involved the most significant code change, the edit map size was 9748 bytes, resulting in a P_{tx} of 25.45%. This can be explained by the fact that even though the blink and flash applications are very different, they share approximately 88% of the base OS code ($100 - \mathtt{w} * 100/L_{new}$). In cases 4 and 5, the P_{tx} values achieved are 5.42% and 7.98%, respectively. The smaller edit map sizes, 2042 bytes and 2815 bytes, respectively, for cases 4 and 5, can be attributed to the fact that even though entire functions are removed in the two changes, the LCS-based approach correctly accounts for the shifts in the other functions within the code image.

In scenarios involving minor to medium code changes in TinyOS, in cases 6 and 8, the approach achieves relatively small edit map sizes of 6 and 1121 bytes, respectively, and correspondingly small P_{tx} values of 0.22% and 9.74%, respectively. Case 7 involves a major change, where the `Blink` application is updated to `RadioCountToLeds`. The large edit map size (10241 bytes) and P_{tx} value achieved (88.5%) is due to the difference in size between L_{old} and L_{new}. Specifically, 8876 bytes of new data must be transferred; only 1365 bytes ($10241 - 8876$) are transferred to rebuild the new image. The P_{tx} values in cases 9 and 10 (10.49% and 10.46%, respectively) can again be attributed to the ability of the LCS-based approach to account for function shifts.

Merge Window Optimization: Table 3 summarizes the impact of varying the merge window size on the size of the resulting edit map; the results are plotted in Figure 8. In each graph, the horizontal axis represents the merge window size, and the vertical axis represents the cost of update ($\mathtt{C_{update}}$). Case 1 is unaffected by window size; this is due to the fact that it consists of a single `CMD_WRITE` operation of length one, and there is nothing to merge. In the remaining cases, the minimum $\mathtt{C_{update}}$ value is achieved at or beyond a merge window value of 2. (Encoding a 2 byte `CMD_MOVE_8` costs the same as a 2 byte `CMD_WRITE`.) We observe that case 3 has the highest rate of change as a function of window size, followed by cases 2, 5, 4, and 1; correlating to their overall $\mathtt{C_{update}}$ costs. A larger $\mathtt{C_{update}}$ cost indicates a higher degree of dissimilarity between two program images, as well as higher fragmentation in the LCS segments. The more fragmented the LCS segments, the higher the chances of merging multiple data segments. This makes the *map optimization* strategy more effective for cases with higher $\mathtt{C_{update}}$ values (major code updates).

Image Reconstruction Cost: Table 4 compares the cost of simple reprogramming ($\mathtt{C_r}$) with the cost of incremental reprogramming ($\mathtt{C_i}$), the results of which are illustrated in the bar plot in Figure 9. We observe that the ratio of the cost of simple reprogramming to the cost of incremental reprogramming strongly correlates to the P_{tx} values achieved in Table 2, in spite of the wide variation in the number of bytes of data to be moved (`m`) and written (`w`) in the FRAM. This can be attributed to the fact that the amount of energy expended in transmitting a byte of data is about 16,000,000 times more than the amount of energy expended to write a byte of data in FRAM, and about 350,000 times more than writing a byte of data to flash. Thus, a higher degree of data transmission savings, even at the expense of flash and other external memory manipulation, correlates with a lower energy footprint.

	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6	Case 7	Case 8	Case 9	Case 10
`m` (Bytes)	0	24766	27873	1996	3492	0	1344	7279	7178	7292
`w` (Bytes)	1	1772	4572	830	1389	2	9875	437	530	503
$\mathtt{C_r}$ (Joules)	198.497	201	202.193	198.919	186.258	13.992	60.857	60.783	60.076	59.242
$\mathtt{C_i}$ (Joules)	0.0269	26.221	51.47	10.782	14.864	0.0317	54.073	5.919	6.304	6.194

Table 4: Comparison of simple and incremental reprogramming costs

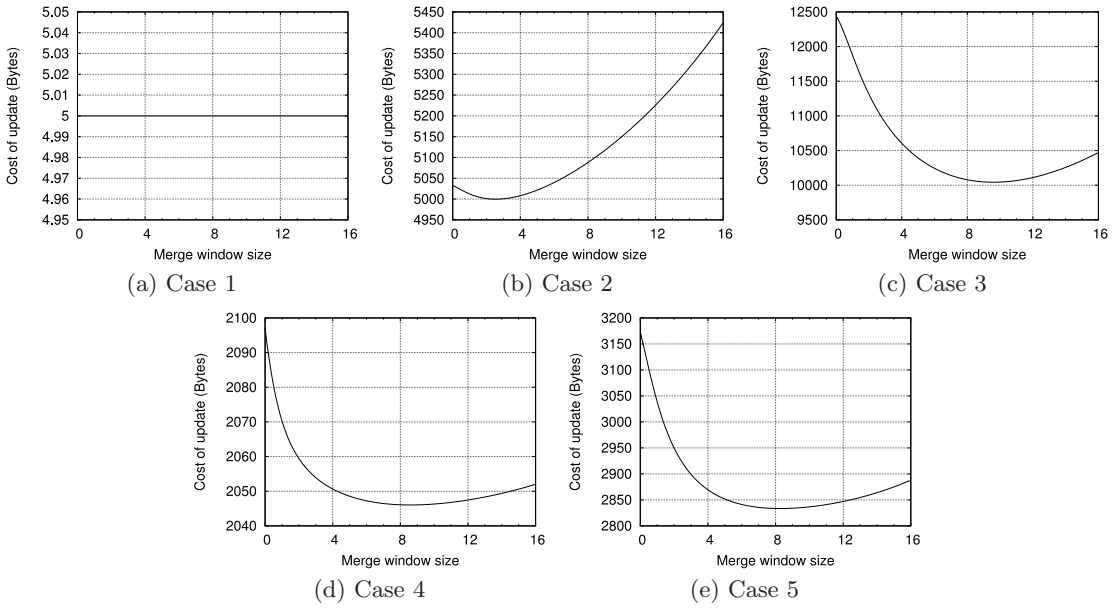

Figure 8: Effects of *map optimization* on C_{update}

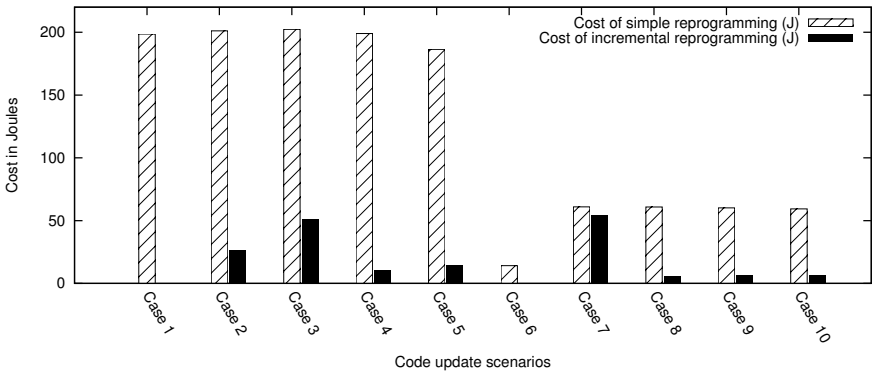

Figure 9: Reprogramming costs for different code upgrade scenarios

Panta et al. report detailed comparisons of delta script sizes of different incremental reprogramming approaches in [17]. Zephyr, when using application-level modifications, reduces the data size to be transmitted to 0.07% for minor changes, and between 1.18% and 23.31% for moderate code changes. Without the application-level modifications, Zephyr reduces the data size to 0.096% for minor code changes, and between 9.17% and 36.63% for moderate code changes. Rsync reduces the size of data to be transmitted to 2.56% for minor code changes, and between 24.47% and 45.65% for moderate changes. In comparison, our approach achieves comparable (and often better) reductions in transmitted data size and significant energy savings for similar code upgrade scenarios. Unlike these approaches, however, our approach does not utilize any knowledge of program code structure. It is platform and language independent.

5. CONCLUSION

We described an incremental code update mechanism for efficient wireless sensor network reprogramming. Our approach uses an adaptation of Hirschberg's algorithm to generate an edit script based on the differences between two program images. We use a heuristic-based optimization strategy to reduce the edit script size, which is then transmitted to sensor nodes using a standard data dissemination protocol. Finally, the sensor nodes decode the edit script and use it to construct the new program image. The approach reduces the data size for transmission to 0.013% for minor changes, and between 5.42% and 13.05% for moderate code changes when using a custom C-based sensor OS. When applied to TinyOS, the approach reduces the data size for transmission to 0.22% for minor changes, and between 9.74% and 10.49% for moderate code changes. Our approach compares favorably to (and often better than) prior work in this area. The reduction in the amount of data needed to be transmitted leads to significant energy savings for wireless sensor network reprogramming. At the same time, our approach is platform and programming language independent, and assumes no knowledge of program code structure.

6. ACKNOWLEDGMENT

This work was supported in part by the National Science Foundation (CNS-0745846, CNS-1126344).

7. REFERENCES

[1] Rone Ilídio da Silva, Virgil Del Duca Almeida, André Marques Poersch, and José Marcos Silva Nogueira. Spatial query processing in wireless sensor network for disaster management. In *Proceedings of the 2nd IFIP conference on Wireless days*, WD'09, pages 194–198, Piscataway, NJ, USA, 2009. IEEE Press.

[2] G. W. Eidson, S. T. Esswein, J. B. Gemmill, Jason O. Hallstrom, T. R. Howard, J. K. Lawrence, Christopher J. Post, C. B. Sawyer, Kuang-C. Wang, and D. L. White. The south carolina digital watershed: End-to-end support for real-time management of water resources. *IJDSN*, 2010, 2010.

[3] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, June 1975.

[4] Jonathan W. Hui and David Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *In Proceedings of the 2nd international*, pages 81–94. ACM Press, 2004.

[5] Digi International. Xbee/xbee-pro oem rf modules - 802.15.4 protocol. ftp://ftp1.digi.com/support/documentation/90000982_A.pdf, 2008.

[6] Jaein Jeong. Node-level representation and system support for network programming. CS294-1 Deeply Embedded Network System Class Project, 2003.

[7] Jaein Jeong and David Culler. Incremental network programming for wireless sensors. In *IEEE Sensor and Ad Hoc Communications and Networks (SECON*, pages 25–33, 2004.

[8] Jaein Jeong, Sukun Kim, and Alan Broad. Network reprogramming. TinyOS document, http://webs.cs.berkeley.edu/tos/tinyos-1.x/doc/NetworkReprogramming.pdf.

[9] Sukun Kim, Shamim Pakzad, David Culler, James Demmel, Gregory Fenves, Steve Glaser, and Martin Turon. Wireless sensor networks for structural health monitoring. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, SenSys '06, pages 427–428, New York, NY, USA, 2006. ACM.

[10] Claudio Lanconelli. Ponyprog serial device programmer. http://www.lancos.com/prog.html.

[11] Hyung Gyu Lee and Naehyuck Chang. Low-energy heterogeneous non-volatile memory systems for mobile systems. *Journal of Low Power Electronics*, 1:52–62, 2005.

[12] Philip Levis and David Culler. Maté: a tiny virtual machine for sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(5):85–95, October 2002.

[13] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, and David Culler. Tinyos: An operating system for sensor networks. In *Ambient Intelligence*. Springer Verlag, 2004.

[14] Philip Levis, Neil Patel, David Culler, and Scott Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI*, pages 15–28, 2004.

[15] Biswajit Mazumder and Jason O. Hallstrom. Sfc: a simple flow control protocol for enabling reliable embedded network systems reprogramming. In *Proceedings of the 50th Annual Southeast Regional Conference*, ACM-SE '12, pages 321–326, New York, NY, USA, 2012. ACM.

[16] Waqaas Munawar, Muhammad Hamad Alizai, Olaf Landsiedel, and Klaus Wehrle. Dynamic tinyos: Modular and transparent incremental code-updates for sensor networks. In *ICC'10*, pages 1–6, 2010.

[17] Rajesh Krishna Panta, Saurabh Bagchi, and Samuel P. Midkiff. Zephyr: efficient incremental reprogramming of sensor nodes using function call indirections and difference computation. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, USENIX'09, pages 32–32, Berkeley, CA, USA, 2009. USENIX Association.

[18] Seon-yeong Park, Dawoon Jung, Jeong-uk Kang, Jin-soo Kim, and Joonwon Lee. Cflru: a replacement algorithm for flash memory. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, CASES '06, pages 234–241, New York, NY, USA, 2006. ACM.

[19] RAMTRON. Fm22l16 datasheet. www.ramtron.com/files/datasheets/FM22L16_ds.pdf, 2010.

[20] Niels Reijers and Koen Langendoen. Efficient code distribution in wireless sensor networks. In *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, WSNA '03, pages 60–67, New York, NY, USA, 2003. ACM.

[21] Victor Shnayder, Mark Hempstead, Bor-rong Chen, Geoff Werner Allen, and Matt Welsh. Simulating the power consumption of large-scale sensor network applications. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, SenSys '04, pages 188–200, New York, NY, USA, 2004. ACM.

[22] Thanos Stathopoulos, John Heidemann, and Deborah Estrin. A remote code update mechanism for wireless sensor networks. Technical report, 2003.

[23] Crossbow Technology. Mote in network programming user reference. TinyOS document, http://webs.cs.berkeley.edu/tos/tinyos-1.x/doc/Xnp.pdf.

[24] Andrew Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, 1999.

[25] Geoff Werner-Allen, Konrad Lorincz, Jeff Johnson, Jonathan Lees, and Matt Welsh. Fidelity and yield in a volcano monitoring sensor network. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 381–396, Berkeley, CA, USA, 2006. USENIX Association.

[26] Tom Yeh, Haru Yamamoto, and Thanos Stathopolous. Over-the-air reprogramming of wireless sensor nodes. In *UCLA EE202A Project Report*, 2003.