Hindawi Publishing Corporation
International Journal of Distributed Sensor Networks
Volume 2014, Article ID 120597, 15 pages
http://dx.doi.org/10.1155/2014/120597

Research Article

EasiLIR: Lightweight Incremental Reprogramming for Sensor Networks

Jiefan Qiu,[1,2] Dong Li,[1] Hailong Shi,[1,2] and Li Cui[1]

[1] *Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China*
[2] *University of Chinese Academy of Sciences, Beijing 100049, China*

Correspondence should be addressed to Dong Li; lidong@ict.ac.cn

Received 3 May 2013; Revised 8 November 2013; Accepted 14 November 2013; Published 16 January 2014

Academic Editor: Yonghe Liu

Energy-efficient wireless reprogramming is key issues for long-lived sensor network. Most of wireless reprogramming approaches focus on the energy efficiency of the data transmission phase. However, the program rebuilding phase on target node is possibly as another significant part of the total reprogramming energy consumption, due to the high energy overhead of reading or writing operation on the energy-hungry nonvolatile memory. In this paper, we propose an energy-efficient reprogramming system—EasiLIR. The core of EasiLIR is to avoid r/w operations on nonvolatile memory as much as possible in two fold. Firstly, we design an *in situ* modification which creates a modified program equivalent to new one without rebuilding program. However, at the cost of no rebuilding program, the redundant binary codes existing in the modified program may break the program time constraint. Therefore, we also design a lightweight segmented rebuilding to directly create the new image in memory. Experiment results show that EasiLIR reduces the r/w operations on nonvolatile memory by approximately 88% and 81% compared to Deluge and R2, and its average reprogramming overhead is about 64.7% of R2.

1. Introduction

Wireless sensor network (WSN) systems may be deployed in inaccessible areas and implement a long-term monitoring task. When the software functionality is changed, wireless reprogramming techniques provide an efficient solution without node redeployment. However, the high-energy overhead prevents the wireless reprogramming from applying in sensor networks. Thus, energy efficiency of wireless reprogramming is essential requirement for the constrained resource sensor node. The reprogramming energy overhead can be coarsely divided into two parts: the communication overhead for wireless transmission or receiving the updating data and the rebuilding overhead for generating new image and storing it.

Prior efforts on reprogramming mainly focus on the former by minimizing the transferred data, such as the incremental reprogramming: Zephyr and Hermes [1, 2] improve the similarity between the old and new versions application by fixing the variable and function addresses; Li et al., and so forth [3], design an update-conscious complier to create the new image according to the old ones for reducing the differences; or Hu et al., and so forth [4], propose a minimal transferred data algorithm by matching the same binary between different versions as much as possible.

On the other hand, most of reprogramming approaches give less consideration to the latter. The traditional view is that rebuilding program happens inside node and its overhead is negligible compared to the communication overhead. However, due to the energy-hungry nonvolatile memory such as EEPROM or flash used for saving binary code, the rebuilding overhead may become one significant part of the reprogramming overhead and even exceeds communication overhead. For this, we propose a lightweight incremental reprogramming system called EasiLIR to reduce the rebuilding overhead.

In EasiLIR, the *in situ* modification is applied instead of the traditional rebuilding. It directly modifies the binary codes stored in memory without entirely rebuilding program. The modified old image is equivalent to the new image. To realize the *in situ* modification, we redefine and substantiate three basic updating commands [1]: *replace*, directly overwrite old binary codes by new; *delete*, delete binary codes by skipping them; *insert*, package inserting binary codes into one function (called *ifunction*) and executed them by invoking *ifunction*. To further save the energy overhead, these *ifunctions* are stored in the low-power volatile memory such as RAM used for saving variable and data.

As the cost of refraining from entirely rebuilding program, the *in situ* modification causes the redundant codes existing in the main program. Due to the impact of the redundant codes on memory space and the execution time, rebuilding program is inevitable.

Therefore, we also design a lightweight segmented rebuilding which rebuild program in each special rebuilding block group (RBG). By this rebuilding, each memory unit is read or written by one time. Moreover, the rebuilding procedure completely occurs in the MCU inside without the participatory of the external component such as the external flash.

To demonstrate the advantage of EasiLIR, we conduct a series of experiments under realistic updating cases. Compared with the other two existing reprogramming system: Deluge and R2, experiment results show the outperformance of EasiLIR with respect to the energy efficiency.

Our contributions in this paper can be summarized in the following.

(1) We present an *in situ* modification mechanism which minimizes the r/w operations on nonvolatile memory and alternatively saves inserting codes in volatile memory. To realize the *in situ* modification, three basis commands are redefined.

(2) We also present a lightweight segmented rebuilding to entirely rebuild program at fewer r/w operations on nonvolatile memory than the traditional rebuilding.

(3) Based on the abovementioned methods, EasiLIR is designed to be a practical energy-efficient approach to meet the requirement of sensor network reprogramming. We conduct experiments with EasiLIR, R2, and Deluge on TinyOS [5] and Ez240 node [6].

The rest of this paper is structured as follows. Section 2 discusses the related work. Section 3 gives the motivation of our work. Section 4 presents the design details. Section 5 describes implementation stages of EasiLIR. Section 6 presents the experiment results. We discuss the limitation of EasiLIR and future work in Section 7. Finally, we conclude our work in Section 8.

2. Related Work

The energy efficiency of wireless reprogramming is key issue in sensor networks. However, the earlier studies such as Deluge [7] give no consideration to energy efficiency. It needs to disseminate the new image and the reprogramming protocol. Stream [8] attempts to reduce the number of transferred bytes by preinstalling the reprogramming protocol in the target node, but it still transmits the new updated program image.

With respect to most of the incremental reprogramming approaches [1–4, 9–11], they pay much attention on how to minimize the communication overhead by merely transmitting the different binary codes between the new and old images. Zephyr [1] and Hermes [2] remain the same reference addresses of functions and variables to improve the similarity between different versions. Li et al. [3] also design a special update-conscious compiler to create the new image program according to the old one by using common register to improve the similarity of different versions. Hu et al. [4] propose the same block-matching algorithm RTMD to reuse the code blocks in the old image as much as possible. The R2 [12] combines the merits both of Zephyr and RTMD. Relative to Zephyr, R2 leverages the relocatable position-independent code to keep the references of functions and variables not being changed between different version. Compared with RTMD, R2 optimizes the *delta* code generation with low time and space complexity.

On the other hand, a few of prior works try to reduce the rebuilding overhead more or less. Koshy and Pandey [11] make a slop region left in the tail of each function in case of inserting code into function, but the size of slop region is hardly determined. Tiny module-linking [9] combine the updating codes with the old function, and then the new function is generated in RAM. However, the new function is also written back into the program flash. Dong et al. [13] utilize low-power RAM to save the updated functions assigned by the developer. It is hard to decide which function will be updated or not when facing a complex sensing application after deployment.

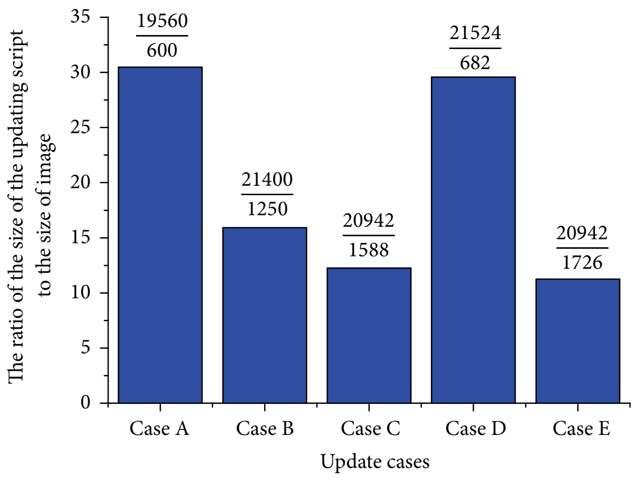
3. Motivation

In this section, we will firstly discuss the necessity of reprogramming and show the traditional rebuilding procedure to explain why the rebuilding overhead is a significant part compared with the communication overhead.

Our WSN system for relic monitoring in Forbidden City Museum [14] has a typical reprogramming demand. After that we first deployed sensor nodes in the museum, some of them were quickly unavailable. We found that the power-supply sink cannot work, because electric power was turned off based on safety considerations during night. The battery-supply Ez240 nodes [6], which cannot receive any response message from the sink, had to continue sending. Their batteries were quickly depleted. When we tried to recall these buggy nodes, we were told that these had been sealed up in the showcase at least 3 months as shown in Figure 1(a). To fix the bug, the wireless reprogramming is the only choice.

However the energy overhead of reprogramming is a bottleneck for applying it. For quantitative analysis of wireless reprogramming, we develop a current-monitoring device [15] (as shown in Figure 6) to measure the duration time and the value of current with respect to different reprogramming operations such as r/w memory or TX/RX. Given the constant voltage, the energy overhead of different operations is shown in Table 1.

(a)



(b)

Figure 1: The sensor network for relic monitoring in Forbidden City: (a) real updating scenarios; (b) the ratio of the size of the updating script to the size of image.

Table 1: Energy overhead (uJ) of different reprogramming operation with 1 kB for Ez240 [6] node.

Operation	Average overhead
Receive data	18902
Read external flash	1015
Read program flash	785
Read RAM	<50
Transmit data (0 dBm)	16590
Write external flash	2458
Write program flash	1850
Write RAM	126

*A memory system of sensor node consists of RAM, the program flash, and the external flash. RAM is used to reserve globe and static variable (.bss and.data section), and the program flash is used for main program (.text section). Both of them are inside Microcontrol Unit (MCU). The external flash is outside and connects with MCU by serial port.

For the incremental reprogramming [1, 2, 4, 12], they locally rebuild program in local sensor nodes, and their rebuilding procedure can be summarized as follows.

Step 1. Sensor node downloads an updating script (also called *delta* script for incremental reprogramming) into the external flash. The updating script contains the updating codes (the changed codes between different versions) and the updating commands.

Step 2. Sensor node runs the commands to combine the updating codes with old image and create a new version image (>15 kB) within the external flash.

Step 3. Write the new image back into the program flash inside MCU and overwrite the old image.

Step 4. Finally reboot target node.

In Step 1, the updating script is written into the energy-hungry external flash, but its size less exceeds 2.5 kB by incremental reprogramming. In Step 2, parts of old image and updating codes are moved into a new space according to the updating commands. This moving causes a large number of r/w operations on the external flash. The number of the operations almost equals the size of new image (>15 kB). In Step 3, sensor node transfers the generated new image (at least 15 kB for TinyOS) from the external flash to the program flash. It also causes a number of flash operations. Summarily, the number of r/w operations on nonvolatile memory is almost 10 times than transmitted bytes. Thus, although rebuilding overhead per byte is almost 10% of the communication as shown in Table 1, yet the rebuilding overhead cannot be neglected.

In practice, we often update software of sensor node with a minor modification such as debug or patch. For our museum application, the size of the updating script transmitted is far smaller than the size of the EzMonitor a sensor node software for Forbidden City Museum. We record the size of the updating script for the different versions of EzMonitor as shown in Figure 1(b). These updating cases will be detailed in Section 6.1.

Figure 1(b) shows the ratio of the size of the updating script to the size of the new image in Step 3 which is written into the program flash using a traditional reprogramming [12]. On average, more than 1265 bytes are transmitted in the five cases. For instance, in updating case D, the transmitted data is up to maximum 1870 bytes (1726 bytes updating script and 144 bytes package header). Meanwhile, the approximate 20 kB data written into the program flash are one order of magnitude more than the transmitted data. The communication totally consumes 69.4 mJ. In contrast, rebuilding program totally consumes 149.5 mJ by the incremental reprogramming [12]. This result reveals the problem that the energy-inefficiency rebuilding still causes much energy waste.

Table 2: Summary of various blocks.

Block name	Description
LCS block	Common block in the largest common sequence
Common block	The successive same binary codes between the new and old version image
Un-LCS block	The successive binary codes between two adjacent LCS blocks in the new version image
Updating block	The successive binary codes between two adjacent LCS blocks in the old version program
Replaced/deleted/inserted/moved block	The successive binary codes which are operated by replace/delete/insert/move commands

4. Design of EasiLIR

In this section, we present the *in situ* modification and then the lightweight segmented rebuilding.

4.1. In Situ Modification. The nature of the *in situ* modification is to reuse old binary codes reserved within the program flash. That is, the common codes between different version image can be directly executed without moving. Because Microcontrol Unit (MCU) sequentially implements codes from low address to high address within a function (C language) or a method (C++/Java language), *in situ* reusing requires as many binary codes within the same sequence as possible.

From my learned knowledge, matching most common codes within same sequence can be translated into a longest common subsequence (LCS) problem which is derived from the early DNA sequence matching in genomics [16]. For example, as shown in Figure 3, old block sequence is $\langle m, \mathrm{A}, n, \mathrm{B}, k, \mathrm{C}\rangle$, and new block sequence is $\langle h, \mathrm{B}, i, \mathrm{C}, j, \mathrm{A}\rangle$. Each common block in the longest common subsequence is defined as one *LCS block* as shown in Table 2. The LCS blocks B and C make up a longest common subsequence. While block A is out of the LCS, both images have it. We define such block as *common block*. We employ the dynamic programming algorithm to solve the LCS problem. This algorithm is in $\mathrm{O}(n^2)$ time, but it totally runs on the host computer and gives less impact on the sensor network.

We define the successive binary codes between two adjacent LCS blocks as *un-LCS block* with respect to *old* image and as *updating block* with respect to *new* image as shown in Table 2. The un-LCS blocks as part of the old image should be modified. The updating blocks as part of the new image are contained in the updating script and transferred into target node. The *in situ* modification is actually to replace all un-LCS blocks with corresponding updating blocks. For this, we needs redefine and substantiate three basic updating commands.

Replace: directly overwrite the old binary codes with the new ones within the program flash.

Delete: skip some useless binary codes by adding one jmp instruction.

Insert: package the inserting binary codes into a special function called *ifunction*, and place one call instruction placed in the insert point. These inserting codes are executed by invoking *ifunction*.

As shown in Figure 2(a), if the size of one un-LCS block C between the LCS blocks A and B is larger than the size of one updating block, the un-LCS block is modified by *replace* and *delete* commands. The updating block transferred is directly written into the main program. The remaining useless codes in block C are skipped by *delete* command adding one jmp instruction.

As shown in Figure 2(b), if the size of the un-LCS block C between the LCS block A and B is smaller than the size of the updating block, the un-LCS block is modified by *replace* and *insert* commands. We separate this updating block into two parts. One part as a replaced block is still directly written into the program flash by *replace* command; the other part which is taken as an inserted block is packaged in an *ifunction* and saved in other place to avoid covering the available block B.

The *ifunctions* should be far from the main program. Considering the energy overhead gap between RAM and the program flash, we put the *ifunction* into RAM. Node runs codes in the inserted block by invoking the *ifunction*.

Beside the above three basic modification commands, in order to reduce the size of the updating script, we also utilize *move* command [1] which directly move the common blocks into a new place unnecessary to retransmit them.

The generated updating script is shown in Algorithm 3, it contains the updating codes and the corresponding updating commands.

The `A1, A2`, respectively, denote the starting addresses of replaced and deleted blocks in the old image. By `INSERT, code2` is packaged in *ifunction* placed in RAM, and one call instruction is written into the inserted point `A2`. By `MOVE`, the `S4` bytes common block located in `O1` is moved into the new position `N1`.

After the target node downloads the updating script, EasiLIR runtime system runs the *in situ* modification described in Algorithm 1.

In line 1, *IFpt* points to an *area* in RAM. By increasing *IFpt* each *ifunction* is sequentially placed in RAM. Line 4–13, these updating commands of the updating script are executed by EasiLIR runtime system.

4.2. Lightweight Segmented Rebuilding. Within the modified main program, the deleted blocks are skipped but still exist in the program flash. With several times of modification, lots of useless fragments finally influence the program time constraint. Thus, entirely rebuilding program is necessary. As shown in Section 3, the traditional rebuilding procedure generates the new image in the external flash and needs to

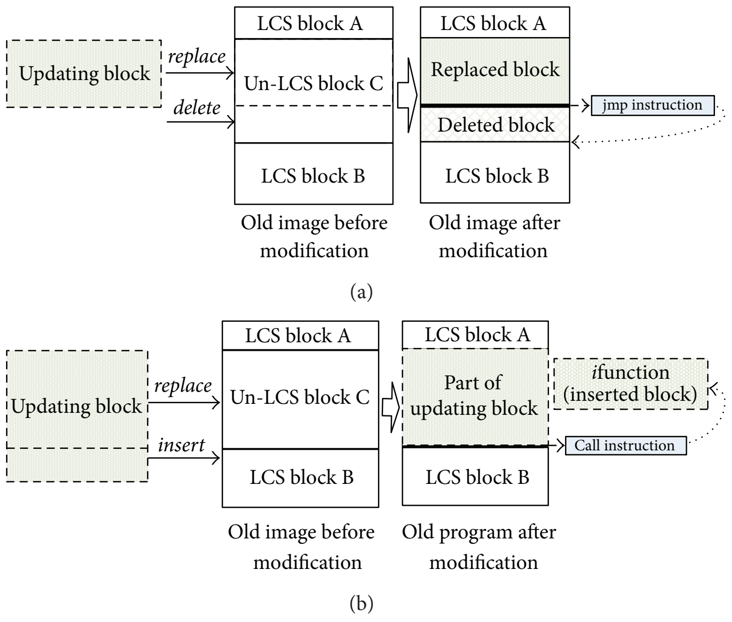


Figure 2: Modify program by *replace*, *delete*, and *insert* commands. (a) *replace* + *insert*: deleting codes by jmp instruction. (b) *replace* + *insert*: executing the inserting codes by calling *ifunction*.

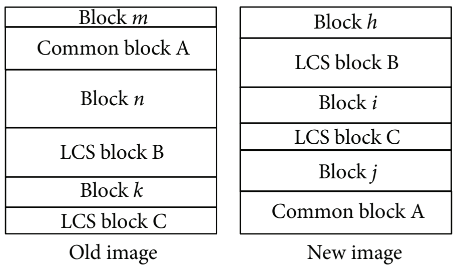


Figure 3: Common block and LCS block: common block B, and C locate in a LCS, and common block A is out of the LCS.

write it into the program flash. Due to the a large number of r/w operations on flash, this method is not energy-efficient.

We propose a lightweight segmented rebuilding to realize inserting all the *ifunctions* and removing all the deleted blocks by moving each LCS block under one operation based on the following two assumptions.

Assumption 1. According to the starting address of deleted block and the inserted point address from low to high, the number of bytes of the deleted blocks may exceed the number of bytes of the inserted blocks.

Assumption 2. The program flash provides enough space to accommodate the new image.

The lightweight rebuilding divides the main program into several segments named rebuilding block groups (*RBG*). In each RBG, it requires that the total number of bytes of deleted blocks exceeds the total number of bytes of inserted blocks based on Assumption 1. Within each RBG, all LCS blocks are moved under one operation to automatically remove all deleted blocks as well as make enough space for all inserted blocks. We give the following explanation of how it works.

Given that in top S of un-LCS blocks (including the deleted and inserted blocks), the total number of bytes of the inserted blocks always exceeds the total number of bytes of the deleted blocks, according to the starting address of the deleted block and inserted point address from low to high:

$$\sum_{j=1}^{S-m} \text{SizeDel}(j) < \sum_{i=1}^{m} \text{SizeIns}(i), \quad (m < S), \tag{1}$$

where m and $S-m$, respectively, denote the number of inserted and deleted blocks in the top S blocks. SizeDel and SizeIns, respectively, denote the number of bytes of each deleted and inserted block. Assume that $(S+1)$th block is the deleted block and has

$$\sum_{j=1}^{S-m} \text{SizeDel}(j) + \text{SizeDel}(S-m+1) \geq \sum_{i=1}^{m} \text{SizeIns}(i). \tag{2}$$

The $(S+1)$ un-LCS blocks and the LCS blocks among the $(S+1)$ un-LCS blocks are made up of the current RBG. The $(S+1)$ th un-LCS block is the RBG point and separates the current RBG from the next RBG.

On the other side, we expand formula (1):

$$\sum_{j=1}^{S-m-n} \text{SizeDel}(j) + \sum_{j=S-m-n+1}^{S-m} \text{SizeDel}(j) + \text{SizeDel}(S-m+1) \geq \sum_{i=1}^{m-1} \text{SizeIns}(i) - \text{SizeIns}(m). \tag{3}$$

```
Input: the m size updating script and the old image
Output: the modified program image;
(1)  Initialization IFpt;
(2)  foreach the updating command C in the updating script;
(3)      switch C
              /* NewAddr/OldAddr, StAddr, InsAddr see in Algorithm 3 */
(4)      case REPLACE:
(5)           overwrite codes located in StAddr with the new one;
(6)      case MOVE:
(7)           move the common block form OldAddr into NewAddr;
(8)      case DELETE:
(9)            write "jmp $ + size" into StAddr;
(10)     case INSERT:
(11)          store the ifunction in RAM with IFpt;
(12)          write "call # IFpt" into InsAddr;
(13)          IFpt = IFpt + size;
(14) end
```

Algorithm 1: *In situ* modification.

The $(S-n)$th un-LCS block owns the highest address among all inserted blocks within the current RBG. It is clear that the $(n-1)$ un-LCS blocks from Sth to $(S-n+1)$th are deleted blocks. The top $(S-n-1)$ un-LCS blocks are satisfied from (1):

$$\sum_{j=1}^{S-n-1-(m-1)} \text{SizeDel}(j) < \sum_{i=1}^{m-1} \text{SizeIns}(i). \tag{4}$$

From formulas (3) and (4), we have

$$\sum_{j=S-m-n+1}^{S-m} \text{SizeDel}(j) + \text{SizeDel}(S+1) \geq \text{SizeIns}(m). \tag{5}$$

Formula (5) indicates that there is enough space to insert the $(S-n)$ th un-LCS block (the m th inserted block) through moving the LCS blocks located among the $(n-1)$ deleted blocks. The new address of the each LCS block, which is located among the $(S-l+1)$th and $(S-l)$th $(l=0,1,\ldots,n)$ un-LCS blocks, is:

$$\$ + \sum_{i=1}^{m} \text{SizeIns}(i) - \sum_{j=1}^{S-m-l} \text{SizeDel}(j), \quad (l = 0, 1, 2, \ldots, n). \tag{6}$$

In (6), \$ denotes the original address of the each LCS block.

Note that a new deleted block next to the $(S-n-1)$th un-LCS block is generated by moving the LCS blocks. The size of the new deleted block is

$$\sum_{i=1}^{m} \text{SizeIns}(i) - \sum_{j=1}^{S-m} \text{SizeDel}(j) - \text{SizeIns}(m) + \sum_{j=S-m-n+1}^{S-m} \text{SizeDel}(j). \tag{7}$$

The remaining $S-n-1$ un-LCS blocks contain $m-1$ inserted blocks and $S-n-m$ deleted blocks. The number of bytes of the remaining deleted blocks equals the number of bytes of the remaining inserted blocks:

$$\begin{aligned}\sum_{i=1}^{m-1} \text{SizeIns}(i) &= \sum_{i=1}^{m} \text{SizeIns}(i) - \sum_{j=1}^{S-m} \text{SizeDel}(j) - \text{SizeIns}(m) \\ &\quad + \sum_{j=S-m-n+1}^{S-m} \text{SizeDel}(j) + \sum_{j=1}^{S-n-m} \text{SizeDel}(j).\end{aligned} \tag{8}$$

Given formulas (8) and (2), there exists enough space for the next inserted block.

By parity of reasoning, the *remaining* inserted blocks in the RBG can be inserted. Therefore, once the LCS blocks in the RBG are moved in the new addresses, they will never be moved again.

After dealing with one RBG, we continue to find the next RBG point and repeat the above process until no RBG point is found anymore (no RBG existing). In terms of Assumption 2, there must be a free space in program flash. Hence we take the free space as last RBG point. Even if the number of bytes of the deleted blocks never exceeds the number of bytes of the inserted blocks, we also take all blocks in the main program as one RBG and the free space as the last RBG point.

We do not mention the moved and replaced blocks alone with the corresponding *move* and *replace* commands in the above explanation. In fact, both kinds of commands have been carried out before rebuilding, so both kinds of blocks will be taken as LCS blocks in the process of rebuilding procedure.

From the above discussion, we design the segmented rebuilding algorithm (see Algorithm 2).

```
Input: the m size updating script and the n size old image;
Ouput: the rebuilt new image
(1) SumDel = SumIns = 0; //sum bytes of del. and ins. blocks
(2) foreach updating Command C in the updating script
(3)         if C is DELETE;
(4)                     SumDel+= size; // the size see in Algorithm 3
(5)         elseif C is INSERT;
(6)                     SumIns+= size;
(8)         end
(9)         if SumDel ≥SumIns;
(10)                set the current deleted block as nowRBGPt;
                    /* the LCS blocks and inserted blocks are between
                    newRGPpt and last one;*/
(11)                move all the LCS blocks to new addresses given in Formula (6);
(12)                    insert all the inserted blocks;
(13)                    set nowRBGPt as lastRBGPt;
(14)         end;
(15) end;
```

Algorithm 2: Lightweight segmented rebuilding.

```
REPLACE · <StAddr=A1> · <size=S1> · <code1>   (1)
DELETE · <StAddr=A2> · <size=S2>   (2)
INSERT · <InsAddr=A3> · <size=S3> · <code2>   (3)
MOVE · <NewAddr=N1> · <OldAddr=O1> · <size=S4>   (4)
```

Algorithm 3: Snapshot of the updating script.

As mentioned above, before rebuilding the program, we handle all the replaced and moved blocks which has been taken as LCS blocks when rebuilding. During this algorithm running, each LCS and inserted block are operated once time. Obviously, from INPUT part, the algorithm requires nearly $m + n$ operations on program flash. Comparing with traditional rebuilding, it requires only almost a half of $m + 2n$ ($n \gg m$) operations on the external/program flash, not to mention the large energy overhead gap of different types of flash.

5. Implementation of EasiLIR

In this section, we firstly give an overview of the implementation of EasiLIR. And then we discuss how to increase the number of LCS blocks, as well as how to improve the similarity between versions for entirely rebuilding.

5.1. The Overview of Implementation. EasiLIR runs on the host computer and the target nodes as shown in Figure 4. At the host computer, the updating script is generated. In order to increase the number of LCS blocks, the sequence of functions in the new version needs to be adjusted according to an old sequence. We also revise each call instruction for improving the similarity between different versions. Then we utilize the dynamic programming algorithm to generate LCS blocks as the updating codes alone with the correspondingly updating command. Finally, the host computer will check whether it is needed to rebuild program based on the program time constraint and the total number of bytes of inserted blocks.

Once the target node detects that reprogramming event happens, it downloads the updating script into the external program. Then it *in situ* modifies the image, or else entirely rebuilding program. At last, program pointer jumps to the main() function to reboot target node.

5.2. Function Rearrangement. In order to increase the number of LCS blocks, we hope to keep more common blocks in the longest common sequence. The functions with same name between different version images always have similar content, especially in the embedded operation system such as TinyOS [5], because functions are named after the low-level components and relevant to their functionality.

Therefore we change the compiling stage of the object file generation process to adjust the sequence of functions according to the last version image. By adjusting, the same name functions in different version images have the same sequence.

5.3. Revising Function Call Instruction. The functions existing in different version images always may be placed in different position. Given entirely rebuilding program, we have to revise the function entry in each call instruction. To modify the entries in all call instructions will bring up an amount of communication overhead. EasiLIR adopts the function call indirections to invoke functions which had been introduced in the literature [1].

We need to revise the call instructions in the link stage. Each call instruction does not store the real function entry, but the fixed address of an item of the indirection table. Each item of the indirection table includes one call instruction with the real function entry and return instruction. Actually calls

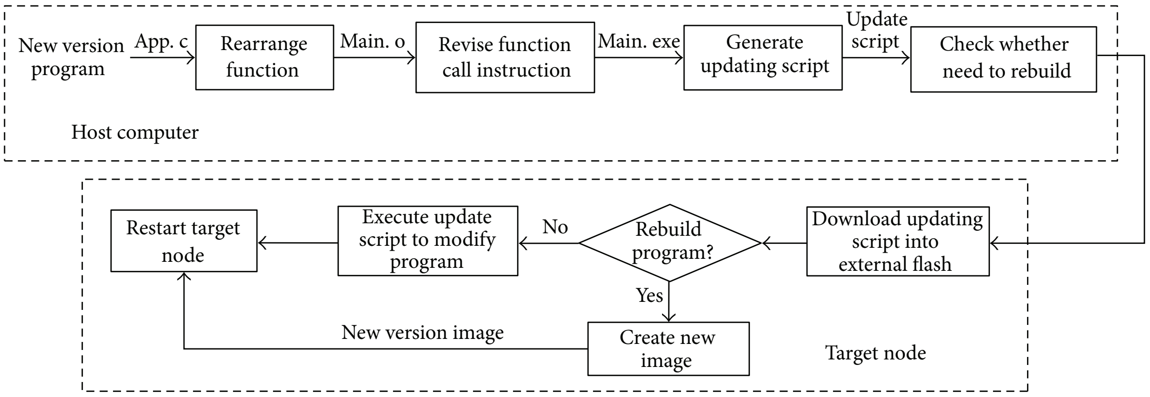


Figure 4: Implementation diagram of EasiLIR.

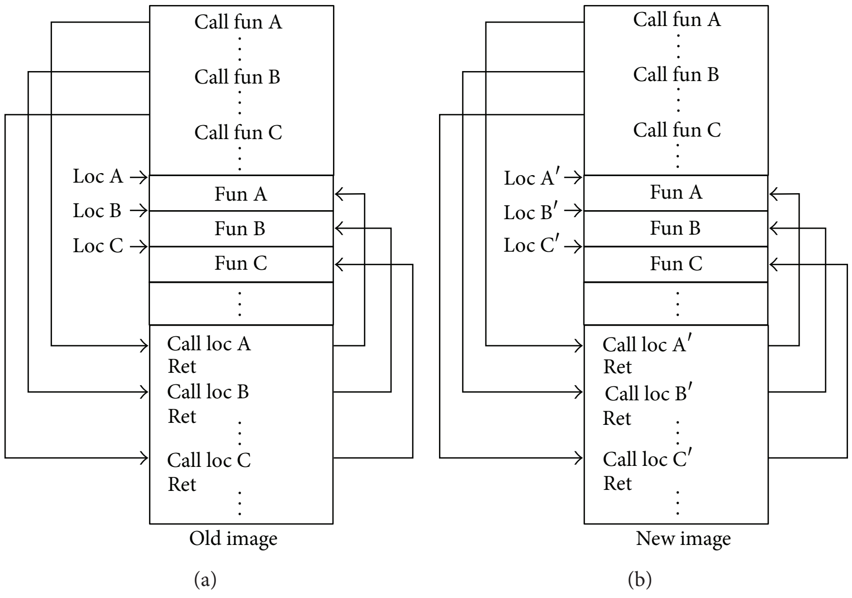


Figure 5: Function call indirection: (a) old image; (b) new image.

to function are made, and when these calls return, the table directs the flow of control back to the line following the call instruction as shown in Figure 5. When entirely rebuilding program, we just revise the indirection table instead of each call instruction in the new image.

6. Evaluation

In this section, we conduct a series of experiments about the single and continuous updating cases. We firstly introduce the experiment methodology and then evaluate the performance of EasiLIR.

6.1. Methodology. We implement and evaluate EasiLIR on our Ez240 node [6] as shown in Figure 6(a). Ez240 is equipped with a 802.15.4-compliant CC2420 [17] radio and MSP430F1611 [18] which has 10 kB RAM (address: 0x1100-0x38FF) and 48 kB program flash (address: 0x4000-0xFFFF). We measure the energy consumption by a current-monitoring device equipped with the current-sense amplifiers MAX9928 [15] as shown in Figure 6(b).

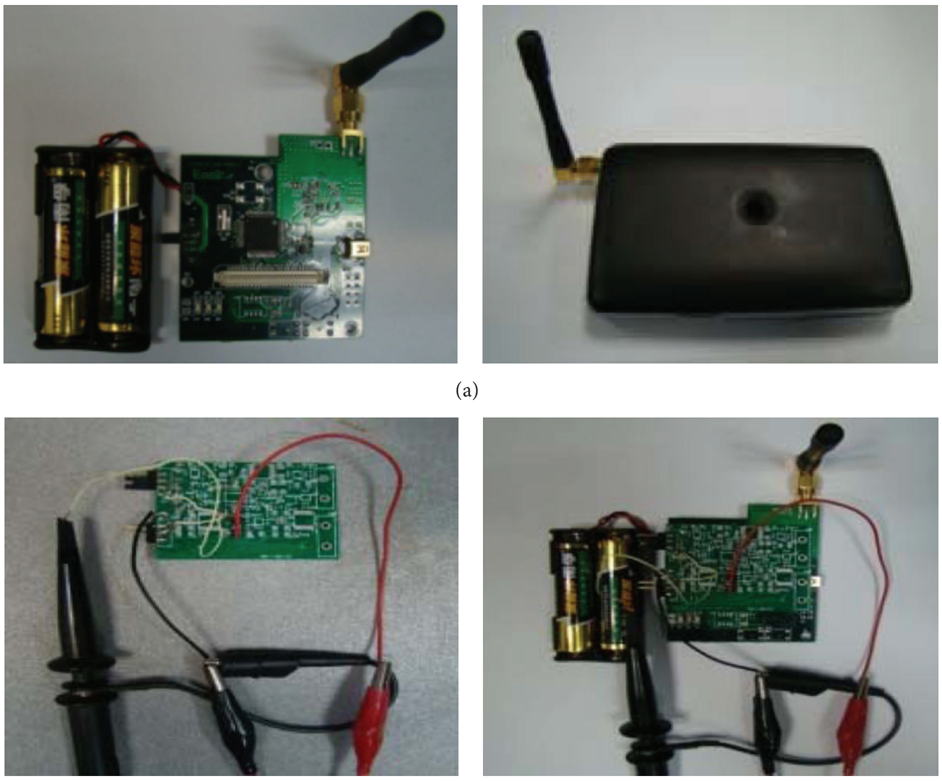
6.1.1. Experimental Setup

(1) The memory space of EasiLIR, *i*functions, and runtime system respectively occupy 3 kB RAM and 12 kB program flash.

(2) The size of transmitted package. Each package contains 100 bytes payload and 8 bytes header information.

(3) The rebuilding condition. The total size of *i*functions exceeds the 3 kB, or the program time constraint cannot be satisfied.

We save one copy of the modified old image in the host computer. By the copy, the host computer evaluates the



(a)

(b)

Figure 6: Sensor node and current-monitoring circuit: (a) Ez240 sensor node and (b) MAX9928 current-monitoring circuit.

number of bytes of the inserted blocks and the execution time and decides whether to rebuild program.

6.1.2. Case Introduction.

(1) *Single Updating Case*. The target node experiences once time reprogramming. We burn an old image into the target node by in-system-programming way in each case:

(i) case 1: delete two lines in Blink to close one LED.

(ii) case 2: modify several lines in Blink to speed up the flashing of LEDs.

(iii) case 3: insert an if-else statement into `main()` function.

(iv) case 4: insert a pattern control function in order to change the flashing pattern of LEDs

(v) case 5: replace Blink with RadioCountToLED. In this case, the total number of bytes of the inserted blocks exceeds 3 kB.

(2) *Continuous Updating Case*. The target node experiences multiple reprogramming. The modified program will be taken as old image in the next round of reprogramming. We design five cases with respect to updating EzMonitor v0.1~0.21:

(i) case A: update EzMonitor from v0.1 to v0.11. In this case, we add new functionality which buffers a sensing data in RAM.

(ii) case B: update EzMonitor from v0.1 to v0.13. In this case, we address the bug that some Ez240 nodes continue to send data to sink during night discussed in Section 3.

(iii) case C: update EzMonitor from v0.11 to v0.2 derived from case A. In this case, we add a load balance mechanism to check battery voltage and optimize the sleep policy. In v0.11, the sleep time of sensor node is fixed. Some nodes have to afford heavy load result in quickly battery depleted. In v0.2, sleep time is dynamically changed according to the current battery voltage.

(iv) case D: update EzMonitor from v0.2 to v0.21 derived from case C. In this case, we remove the sensing function from some Ez240 nodes which just forward data as routing node. The host computer finds that the implementation time cannot be satisfied and asks EasiLIR runtime system to run the entire rebuilding process.

(v) case E: update EzMonitor from v0.13 to v0.2 derived from case B. The entire rebuilding program happens because the total size of *ifunctions* exceeds 3 kB.

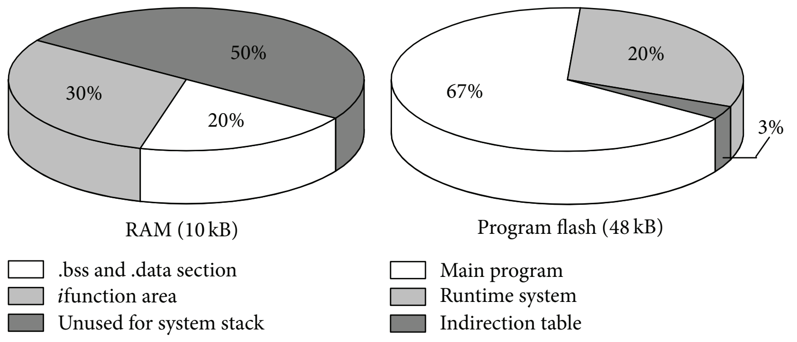


FIGURE 7: Each component in memory layout.

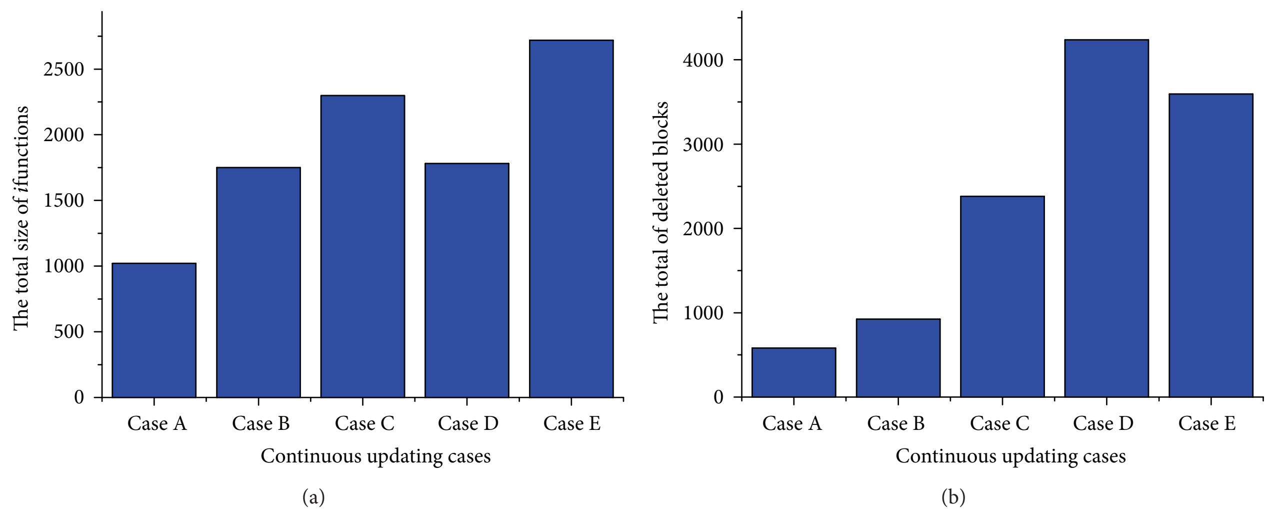


FIGURE 8: Size of updating blocks: (a) *i*functions and (b) deleted blocks.

6.2. Memory Overhead. As shown in Figure 7, the program flash stores the main program (.text section), EasiLIR runtime system, and the indirection table. RAM stores the globe or static variables (.bss and .data section) and *ifunctions*.

With respect to RAM, the size of .bss and .data section hardly exceeds 1.5 kB for TinyOS. In EzMonitor v0.11, an array for sensor data causes the.bss section up to maximum 1326 bytes among all updating cases. Meanwhile, the total size of the inserted blocks (*ifunction*) is about 1 kB~2.5 kB in the continuous cases as shown in Figure 8(a).

With respect to the program flash, the size of EasiLIR runtime system has independent TX/RX and r/w flash functionality up to 12 kB. With respect to the indirection, each item occupies 4 bytes space. The main program contains about 150–300 functions, so it is reasonable to give 1.4 kB (>4 bytes∗300) space for the indirection table.

Although the deleted blocks are skipped, they still exist in the main program and occupy the rare memory space. Without rebuilding program, Figure 8(b) gives the total size of the deleted blocks in each continuous case. The total size of the deleted blocks increases along with modification times. In case D, the total size of the deleted blocks arrives at 4238 bytes (62 deleted blocks).

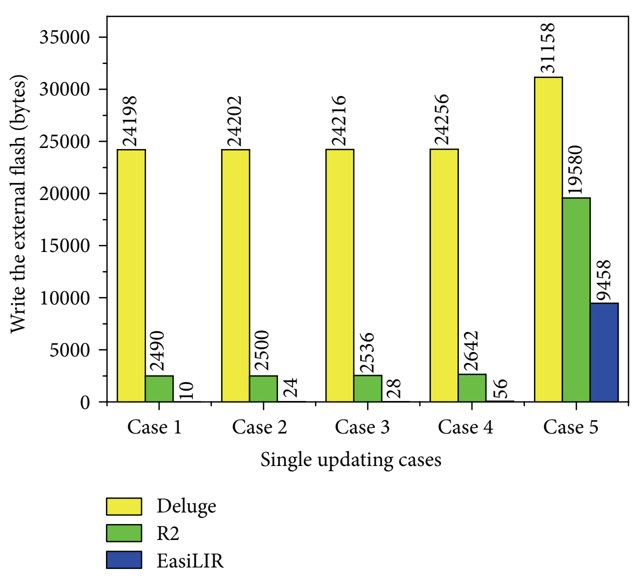
6.3. Transmitted Data. For illustrating the impact of communication overhead on reprogramming, we compare the EasiLIR with R2 [12] and Deluge [7]. Table 3 gives the number of the transmitted data.

Obviously, in all the updating cases, EasiLIR and R2 have less transmitted data than Deluge. The updating script of Deluge contains an entire image as well as a reprogramming protocol, and its average size script exceeds 32 kB, since a minor modification may bring up a large number of transmitted data. For example, the modifying codes account for about 0.07% of the updating script in case 1. R2 adopts the incremental mechanism.

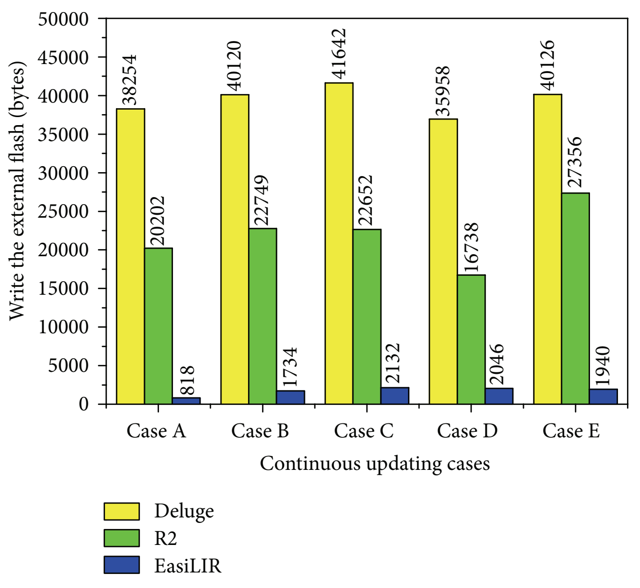
The updating script only contains the *changed* codes between versions and the updating commands. The size of the script is far smaller than the entire program image.

Comparing with EasiLIR, R2 uses an optimal dynamic algorithm [12] to differentia the changed codes and generate the smaller-size updating script than EasiLIR. So EasiLIR needs to transmit more data than R2 in case 5 and case A–case E.

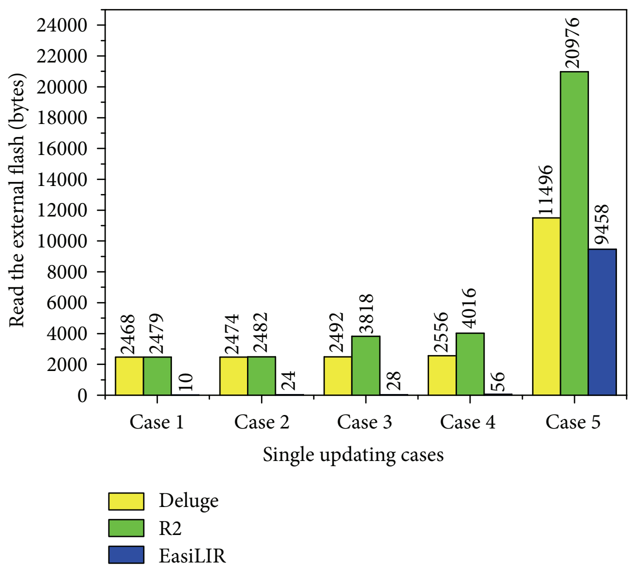
On the other hand, R2's updating script includes the relocation entry in order to rebuild program in every time reprogramming, but it is not necessary for EasiLIR when to



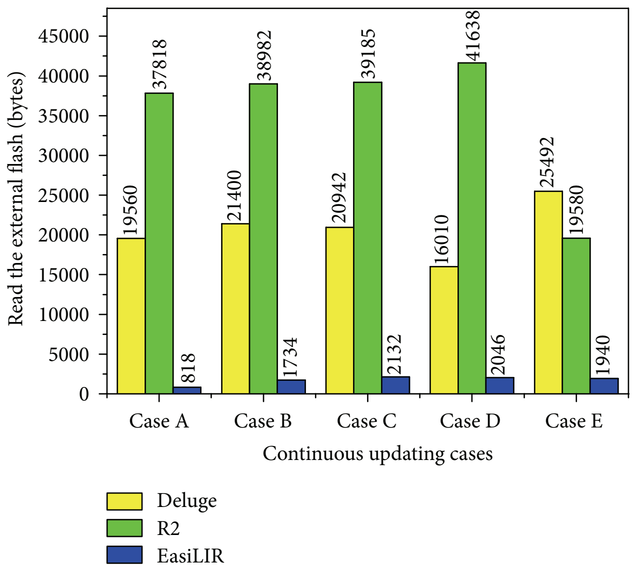
(a)



(b)



(c)



(d)

Figure 9: Operation on the external flash: (a) writing data in the single updating cases; (b) writing data in the continuous updating cases; (c) reading data in the single updating cases; (d) reading data in the continuous updating cases.

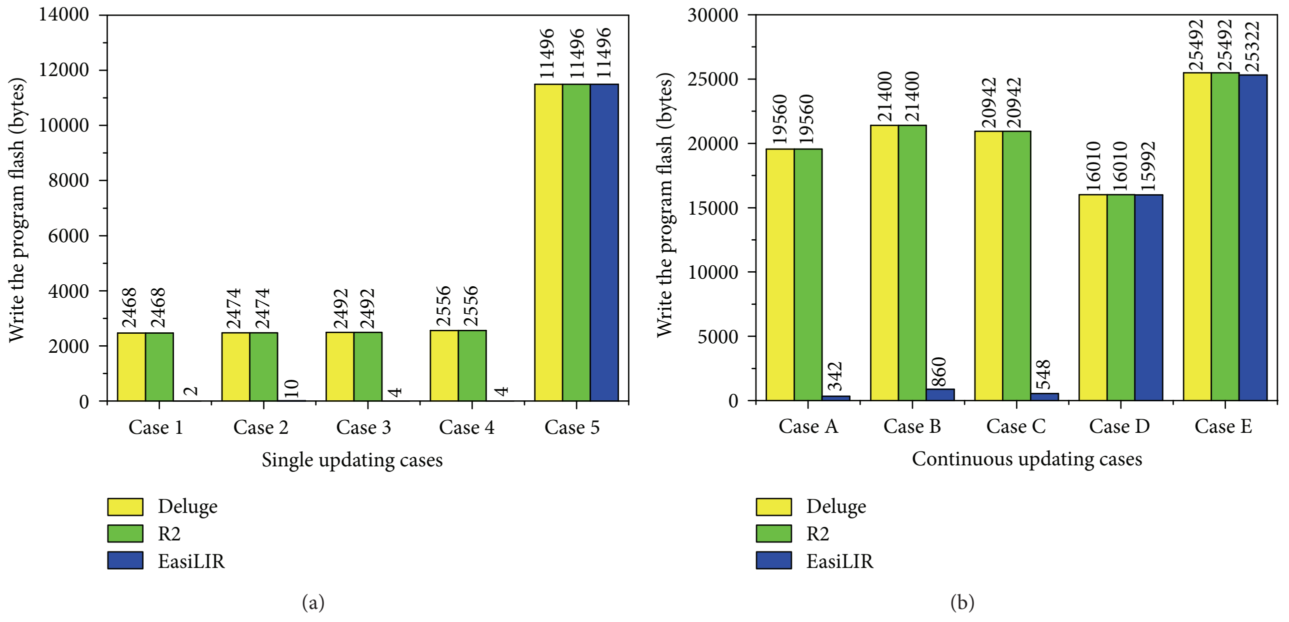
Table 3: The transferred data size with Deluge, Hermes, and EasiLIR (bytes).

	Case 1	Case 2	Case 3	Case 4	Case 5	Case A	Case B	Case C	Case D	Case E
Deluge	24198	24202	24216	24256	31158	38254	40120	41642	36950	41178
R2	22	26	44	86	8084	650	1354	1716	738	1870
EasiLIR	10	24	28	56	9458	818	1734	2132	2046	1940

in situ modify program. Therefore, in case 3 and case 4, the number of the transmitted data of EasiLIR is smaller than R2.

6.4. Read/Write Data. Figures 9 and 10 show the size of the reading/writing data from/to the program/external flash. And then Table 4 gives the comparison of r/w on external/program flash among approaches. We will give the analysis for each case.

Figures 9(a) and 9(b) show the size of the writing data into the external flash. In rebuilding procedure, it is necessary to download the updating script in the external flash. Thus, by Deluge, the size of writing data into the external flash is



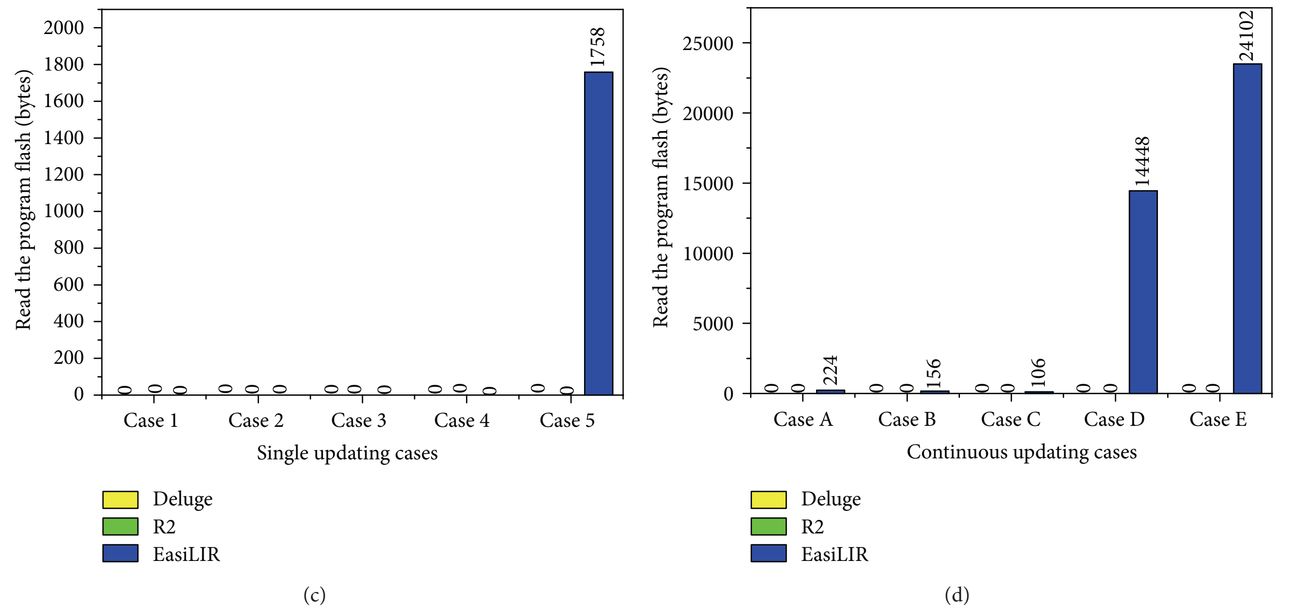


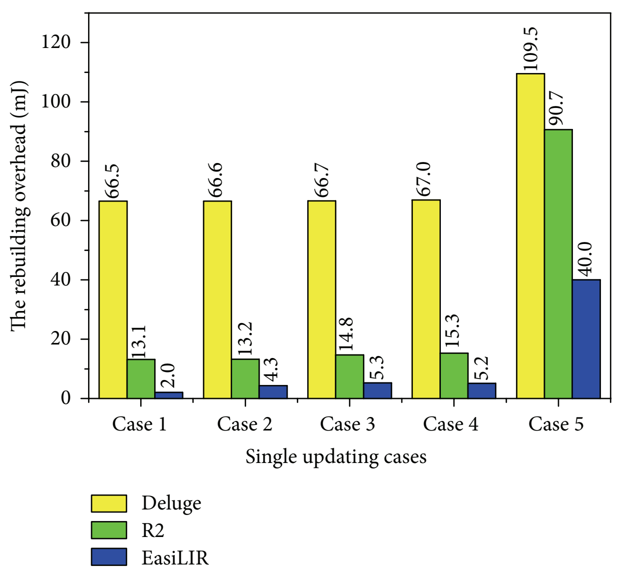
FIGURE 10: Operation on the program flash: (a) writing data in the single updating cases; (b) writing data in the continuous updating cases; (c) reading data in the single updating cases; (d) reading data in the continuous updating cases.

TABLE 4: Comparison of r/w on external/program flash with Deluge, R2, and EasiLIR (bytes/bytes).

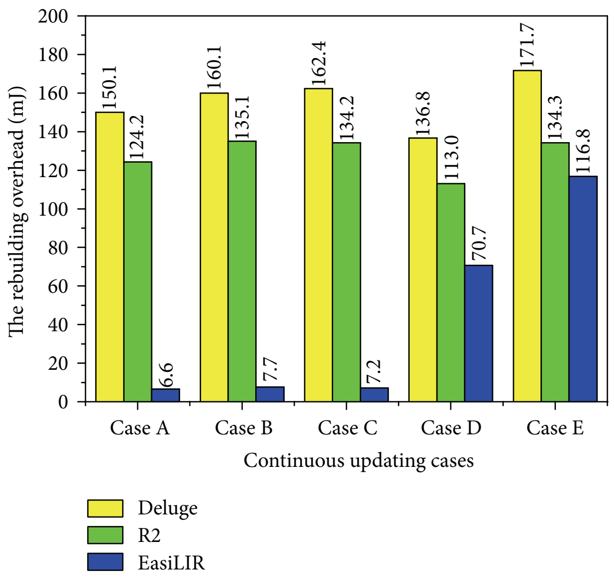
	Case 1	Case 2	Case 3	Case 4	Case 5	Case A	Case B	Case C	Case D	Case E
Deluge/EasiLIR	1324	502	486	253	1.68	36.7	20.7	16.9	1.99	1.67
R2/EasiLIR	337	128	147	79.4	1.61	36.8	20.8	16.8	2.15	1.32

determined by the size of the new image and its protocol. By R2, writing the external program is mainly caused by combining the updating code into the old image except for downloading the updating script. By EasiLIR, the above combining process is replaced by the *in situ* modification and the segmented rebuilding within the program flash, so the size of writing data into the external program is only determined by the updating script.

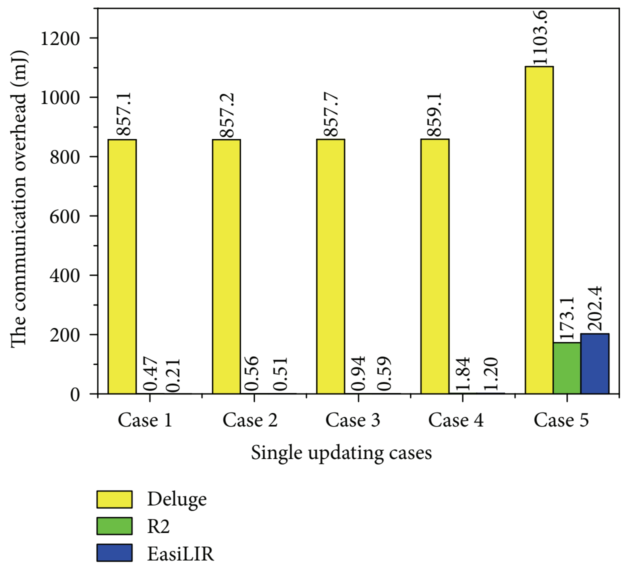
Figures 9(c) and 9(d) show the size of the reading data from the external flash. R2's rebuilding process is not efficient because the updated codes need to be combined with the original codes. Compared with Deluge, the integrated new image is directly moved from the external flash to program flash without combination. By EasiLIR, only the updating script is read from the external flash.



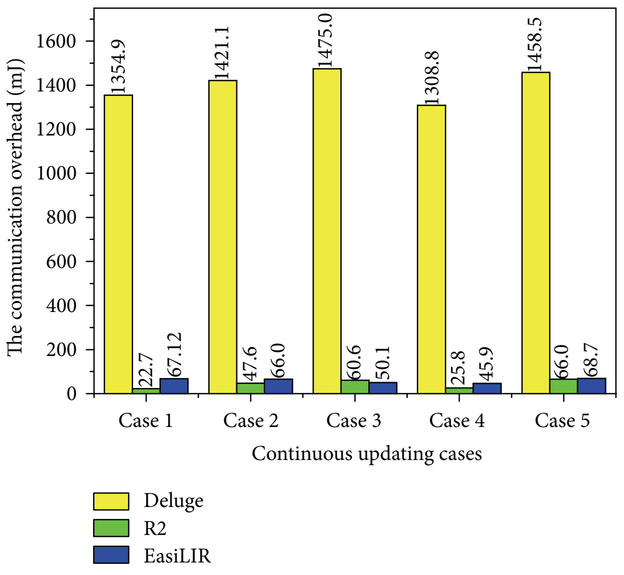
(a)



(b)



(c)



(d)

FIGURE 11: Component of reprogramming overhead: (a) rebuilding overhead in the single updating cases; (b) rebuilding overhead in the continuous updating cases; (c) communication overhead in the single updating cases; (d) communication overhead in the continuous updating cases.

Figures 10(a) and 10(b) show the size of the writing data into the program flash. By Deluge and R2, the new image is written into the program flash; thus the size of the writing data equals the size of the new version image. By EasiLIR, in case 1 and case 2 with a minor modification, a few of codes are written into the program flash. In case 3 and case 4, the inserted codes are actually saved in RAM, and one call instruction (4 bytes) is written into the program flash. In case A–case C, the replaced blocks, jmp, and call instructions are written into the program flash. In case 5, case D, and case E, the EasiLIR runtime system conducts a segmented rebuilding process to create the new image. Because each LCS block in the main program is moved only once, the size of writing data is close to the size of the new image.

Figures 10(c) and 10(d) show the size of the reading data from the program flash. By Deluge and R2, the size of the reading data is zero, because the new image is completely created in the external flash without the participatory of the program flash. By EasiLIR, it is also not necessary to reading any data from the program flash in case 1–case4. In case A–case C, the reading data are caused by the *move* command, which copies common blocks and pastes them in new places. In case 5, case D, and case E, each LCS block is moved only once; thus the size of reading data does not exceed the size of the new image.

Table 5: Comparison of the reprogramming overheads with Deluge, R2, and EasiLIR (mJ/mJ).

	Case 1	Case 2	Case 3	Case 4	Case 5	Case A	Case B	Case C	Case D	Case E
Deluge/EasiLIR	412	191	157	147	5.01	20.41	21.46	28.58	12.40	8.78
R2/EasiLIR	6.10	2.86	2.664	2.72	1.08	1.993	2.47	3.40	1.19	1.08

From the above discussion, it is easy to understand why EasiLIR owns low rebuilding overhead. The reason is that we use the RAM and the program flash to complete the *in situ* modification and the segmented rebuilding instead of the external flash as much as possible.

6.5. Energy Overhead. Figure 11 shows the components of the reprogramming overhead. For straightforward comparison of the rebuilding and communication overhead, we take energy unit (mJ) as a measure of the energy overhead.

Obviously, Deluge has the highest reprogramming overhead due to the largest updating script. By Deluge, the rebuilding overhead is far lower than the communication overhead, and it is hard to say that the rebuilding overhead is a significant part. R2 has the lowest communication overhead among three approaches due to the optimal dynamic algorithm [12], but its average rebuilding overhead is almost 1.97 times than its communication overhead.

Using the *in situ* modification and the lightweight segmented rebuilding, EasiLIR keeps the lowest rebuilding overhead among three approaches under all the cases as shown in Table 5. The EasiLIR's average reprogramming overhead is 80.1% and 58.7% of R2's, respectively, under the single updating cases and the continuous updating cases. Even in case 5, case D, and case E, the segmented rebuilding happens, the EasiLIR's rebuilding overhead is still 33% lower than R2's due to disabling the energy-hungry external flash.

7. Limitation and Future Work

It is unavoidable to introduce the useless fragments after *in situ* modification and these fragments break the program time constraint. Therefore, the *in situ* modification is not suitable for precision timed cases.

On the other hand, the fragments also complicate the structure of the main program and make the updating process more difficult. For instance, *delete* command is to skip deleted codes by writing jmp instruction into main program. If the deleted codes are available in the next round of reprogramming, the updating script needs one extra *replace* commands to revise the jmp instruction.

Therefore, we will explore the partial *in situ* modification to combine some local and adjacent deleted/inserted blocks into the main program but not entirely rebuild program. By the partial *in situ* modification, we need to find the optimal tradeoff between the rebuilding and communication overhead. In future work, we also test the performance of EasiLIR in different microembedded OSes such as Contiki [19] and sensor nodes such as Mica [20] or Telos [21].

8. Conclusion

Most of the prior efforts focus on how to reduce the communication overhead by minimizing transmitted data, but the rebuilding overhead is also a significant part for reprogramming actually. In this paper, we present a lightweight reprogramming system EasiLIR which employs *in situ* modification instead of rebuilding program and largely reduces the rebuilding overhead. At the same time, the *in situ* modification also introduces the useless redundant codes existing in a modified program. We also design and substantiate a lightweight segmented rebuilding to reduce the operation on flash. At last, we conduct a series of experiments in the EZ240 testbed with EzMonitor a practical application based on TinyOS. Experiment results show the EasiLIR advantage of energy efficiency over prior reprogramming system.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

This paper is supported in part by the National Natural Science Foundation of China (NSFC) under Grant no. 61003293, the Beijing Natural Science Foundation (BNSF) under Grant no. 4112054, the "Strategic Priority Research Program" of the Chinese Academy of Sciences under Grant no. XDA06010403, and the International S&T Cooperation Program of China (ISTCP) under Grant No. 2013DFA10690.

References

[1] R. K. Panta, S. Bagchi, and S. P. Midkiff, "Zephyr: efficient incremental reprogramming of sensor nodes using function call indirections and difference computation," in *Proceedings of the USENIX Annual Technical Conference*, 2009.

[2] R. K. Panta and S. Bagchi, "Hermes: fast and energy efficient incremental code updates for wireless sensor networks," in *Proceedings of the 28th IEEE Conference on Computer Communications (INFOCOM '09)*, pp. 639–647, April 2009.

[3] W. Li, Y. Zhang, J. Yang, and J. Zheng, "UCC: update-conscious compilation for energy efficiency in wireless sensor networks," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*, pp. 383–393, June 2007.

[4] J. Hu, C. J. Xue, Y. He, and E. H.-M. Sha, "Reprogramming with minimal transferred data on wireless sensor network," in *Proceedings of the 6th IEEEInternational Conference on Mobile Adhoc and Sensor Systems (MASS '09)*, pp. 160–167, October 2009.

[5] P. Levis, S. Madden, J. Polastre et al., “TinyOS: an operating system for sensor network,” in *Ambient Intelligence*, pp. 115–148, Springer, Berlin, Germany, 2005.

[6] L. Cui, Q. Liu, and D. Li, “The system framework and equipments of the internet of things,” *Communications of the China Computer Federation*, vol. 6, no. 4, pp. 18–22, 2010.

[7] J. W. Hui and D. Culler, “The dynamic behavior of a data dissemination protocol for network programming at scale,” in *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems (SENSYS ’04)*, pp. 81–94, November 2004.

[8] R. K. Panta, I. Khalil, and S. Bagchi, “Stream: low overhead wireless reprogramming for sensor networks,” in *Proceedings of the 26th IEEE International Conference on Computer Communications (INFOCOM ’07)*, pp. 928–936, May 2007.

[9] S.-K. Kim, J.-H. Lee, K. Hur, K.-I. Hwang, and D.-S. Eom, “Tiny module-linking for energy-efficient reprogramming in wireless sensor networks,” *IEEE Transactions on Consumer Electronics*, vol. 55, no. 4, pp. 1914–1920, 2009.

[10] P. Levis, N. Patel, D. Culler, and S. Shenker, “Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks,” in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI ’04)*, 2004.

[11] J. Koshy and R. Pandey, “Remote incremental linking for energy-efficient reprogramming of sensor networks,” in *Proceedings of the 2nd European Workshop onWireless Sensor Networks (EWSN ’05)*, pp. 354–365, February 2005.

[12] W. Dong, Y. Liu, C. Chen, and J. J. Bu, “R2: incremental reprogramming using relocatable codes in networked embedded systems,” in *Proceedings of the 30th IEEE International Conference on Computer Communications (INFOCOM ’11)*, pp. 376–380, 2011.

[13] W. Dong, Y. Liu, X. Wu, L. Gu, and C. Chen, “Elon: enabling efficient and long-term reprogramming for wireless sensor networks,” in *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS ’10)*, pp. 49–60, June 2010.

[14] D. Li, W. Liu, and L. Cui, “EasiDesign: an improved ant colony algorithm for sensor deployment in real sensor network system,” in *Proceedings of the 53rd IEEE Global Communications Conference (GLOBECOM ’10)*, pp. 1–5, December 2010.

[15] “MAX9928 introduction,” http://www.maximintegrated.com/datasheet/index.mvp/id/5753.

[16] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Alogrithm*, 3th edition, 2009.

[17] “CC2420 introduction,” http://www.ti.com/lit/ds/symlink/cc2420.pdf.

[18] “MSP430x1xx Family User’s Guide (Rev. F),” http://focus.ti.com/lit/ug/slau049f/slau049f.pdf.

[19] A. Dunkels, B. Grönvall, and T. Voigt, “Contiki: a lightweight and flexible operating system for tiny networked sensors,” in *Proceedings of the IEEE Workshop on Embedded Networked Sensors (EmNets ’04)*, pp. 455–462, 2004.

[20] “MicaZ introduction,” http://www.xbow.com.

[21] J. Polastre, R. Szewczyk, and D. Culler, “Telos: enabling ultra-low power wireless research,” in *Proceedings of the ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN ’05)*, pp. 364–369, April 2005.