Hindawi
Wireless Communications and Mobile Computing
Volume 2018, Article ID 2940952, 11 pages
https://doi.org/10.1155/2018/2940952

Research Article

RePage: A Novel Over-Air Reprogramming Approach Based on Paging Mechanism Applied in Fog Computing

Jiefan Qiu, Sai Li, and Bin Cao

College of Computer Science, Zhejiang University of Technology, Hangzhou, China

Correspondence should be addressed to Bin Cao; bincao@zjut.edu.cn

Received 26 January 2018; Accepted 25 March 2018; Published 14 May 2018

Academic Editor: Xuyun Zhang

In fog computing, fog nodes running different tasks near the sources of data are required. Limited to on-board resource, fog node finds it hard to execute multiple tasks and needs over-air reprogramming to rearrange them. With respect to reprogramming, energy efficiency is one of the key issues for over-air reprogramming. Most of traditional reprogramming approaches focus on the energy efficiency during data transmission within network. However, program rebuilding on fog node is, as another significant energy cost, caused by writing/reading local high-power memory. We present a novel incremental reprogramming approach, RePage, in three stages. Firstly, we design a function paging mechanism that makes similar functions to one function page and caches them in low-power volatile memory to save energy. Secondly, we design new cache replacement algorithm for function page considering both modification times and range on the page. At last, further reducing writing/reading operations, we also redesign function invocation manner by centralized managing function addresses. Experiment results show that RePage reduces the sum of reading/writing operations on volatile memory by 89.1% and 92.5% compared to EasiCache and Tiny Module-link, and its hit rate is improved by 10.4% to Least Recently Used (LRU) algorithm.

1. Introduction

In some applications of traditional front-end networks such as wireless sensor networks or ad hoc networks, the users are familiar with the deployment environment after a period of data collection and computing by nodes. More effective functionality modules can be developed and installed on the node by over-air reprogramming to optimize the local software.

Moreover, with emergence of fog computing, more and more systems of fog computing paradigm witness success in many different fields. But the most existing fog computing systems are similar to the traditional sensor networks, which are to solve the specific problem or meet the specific requirements. In the construction of such systems, developer adopts different hardware devices, software module, and even programming language. On the other hand, fog computing systems require dealing with date by fog nodes which are nearby data source. It is inevitable that resource-limited fog nodes execute multiple tasks; for example, the nodes deployed in building need to monitor sound wave and detect vibration damage; the limited storage results in them being unable to achieve the above tasks at the same time; and it is wasteful and inefficient to deploy two kinds of nodes for each task. This necessitates frequently modifying executing application or rearranging tasks through over-air reprogramming.

In most cases, the resource-limited fog nodes are powered by battery. Frequently, over-air reprogramming will shorten node's life and becomes the main bottleneck of large-scale application. The energy cost of over-air reprogramming mainly comes from communication cost when transmitting codes and from rebuilding cost when reading and writing the memory cell. Now, most of research works about reprogramming focus on how to reduce the communication cost. For example, incremental reprogramming approaches [1–8] only need to transmit the differences between executing old program image and the new one with less transmission cost than entire new image.

However, most of incremental reprogramming approaches need to rebuild the new image in the local memory of sensor node. The popular sensor node TelosB [9], for instance, owns internal flash + RAM within the MSP430

Table 1: Energy cost (uJ) of r/w operation in different memories with 1 kB for TelosB node [9].

Operation	Average cost
Read external flash	1015
Read program Flash	785
Read RAM	<50
Write external flash	2458
Write program flash	1850
Write RAM	126

series MCU and independent on-board external flash. When rebuilding code, the different codes and update-relevant instructions, which are packaged into *delta* script [3], need to be firstly read in the external flash. Then, executing update-relevant instructions, the different codes are combined with old program image to generate new image. At last, write new program image into internal flash in MCU. As shown in Table 1, read and write (r/w for short) flash has higher energy cost than RAM. Actually, inserting some instructions may bring about sharp increase of rebuilding cost, because each instruction is contiguously stored, and to avoid covering useful origin one, most parts of instructions may be moved in new address by r/w flash and make room for the inserted instructions. That possibly makes rebuilding cost actually close to the communication cost.

With respect to the fog computing applications, several users possibly access deployed node at the same time. Due to existing large-capacity on-board external flash, several programs can be prestored in this flash and switched according to the needs of different users in updating node's software. Also, incremental reprogramming approach makes only storing difference codes in the flash to save memory space. In such cases, the reprogramming cost will largely depend on rebuilding cost.

In our previous work, we have proposed an incremental reprogramming approach named EasiCache [10] for sensor nodes reprogramming. In this approach, frequently changing codes are stored and executed in low-power and limited-capacity volatile memory such as RAM without the participation of nonvolatile memory. EasiCache caches part of codes on the function as unit, and function's size is uncertain. Updating too frequently causes storage fragmentation and wastes storage space. Therefore, we propose a novel incremental reprogramming approach named RePage based on function paging mechanism applied in fog nodes. The core idea of this approach is to organize functions in the form of function page to solve the decline problem of cache usage and improve the hit rate. We study this approach from the following three aspects.

First of all, the uncertain function size leads to the waste of storage space. In RePage, we analyze the invocation relationship among functions and define function similarity degree (FSD). And then, based on FSD, we design one novel function paging mechanism, which makes similar functions being put into one function page and updates them at the same time.

Secondly, we improve the cache usage and hit rate through a novel cache replacement algorithm considering modification times and range of each function page. This replacement algorithm is able to adaptively adjust the impact from modification times and range.

Thirdly, in order to improve the similarity between new and original program images, we redesign function invocation method according to the characteristics of function page. We use the register relative addressing instead of direct addressing and manage the function address at centralized manner. By this, once function pages are moved, the runtime system will not modify each function's entry address.

Finally, in order to verify the effect of the paging mechanism, we conduct a series of experiments. In experiments, the resource-limited fog nodes are continuously updated by reprogramming without redeployment, and we observed the performance of RePage from hit rage, energy cost, and storage cost.

The rest of this paper is structured as follows: Section 2 shows the current problem in caching codes; Section 3 introduces the design and implementation of RePage; Section 4 describes experimental scenarios and evaluates our approach; Section 5 gives related work and Section 6 gives conclusion.

2. Motivation

In our previous study called EasiCache [10], the code segments are cached and executed in the type of function. When old function is replaced, runtime system for EasiCache distributes memory space to store new function. However, after multiple reprogramming processes, it leads to a storage fragmentation problem and eventually decreases cache performance. In order to describe the storage fragmentation problem, we define the fragmentation degree as follows:

$$\text{fragmentation degree} = \frac{\text{free space of cache}}{\text{space of cache}}. \quad (1)$$

We test EasiCache in randomly updating scenario on TelosB node (details shown in Section 4.1). TelosB equips MSP430 series MCU with 10 KB RAM and 48 KB internal flash [9]. The size of cache is set to 8 KB. Figure 1 shows the EasiCache continuous updating cases with 25 times and their fragmentation degrees.

We can clearly see that the cache fragments began to increase after more than 5 times reprogramming. After 23 times, the fragmentation degree has been more than 30%. This means that new functions are hard to be cached in RAM. The only way to eliminate fragments is to restore all of caching functions in new addresses and that inevitably brings about huge r/w operations on flash.

In addition, in the real process of caching codes, the replaced functions will be saved in the original position of internal flash. Actually, EasiCache has learned from the previous literature [11]. If the function size increases during caching, the size-increased function is put into free space of the main program (.text segment) directly to avoid tuning the other functions' position. However, the internal flash capacity is also tight. After several updates, the free space may run out,

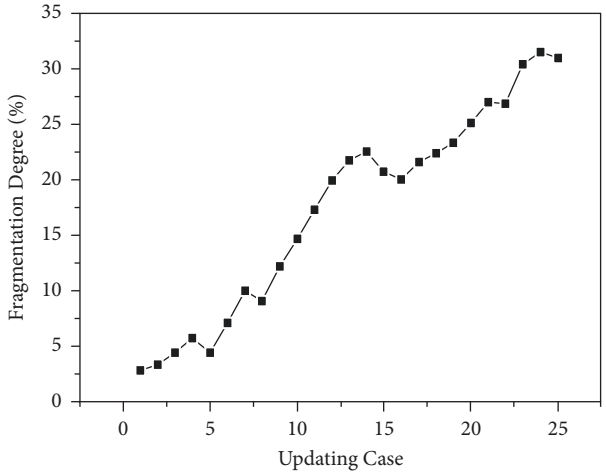

Figure 1: Fragmentation degree in continuously updating cases.

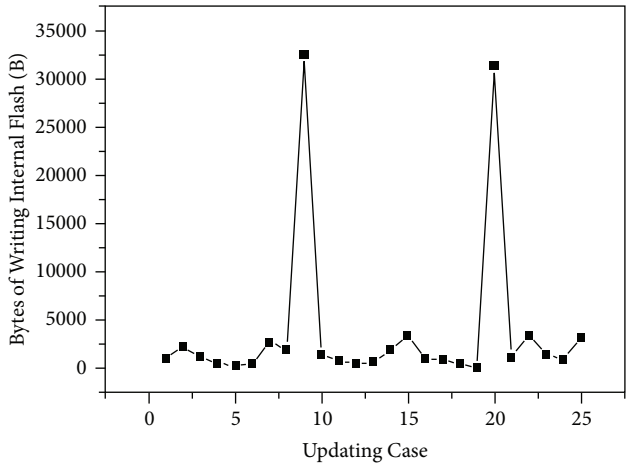

Figure 2: Bytes of writing internal flash in continuously updating cases.

and all functions still need to be rearranged. Figure 2 shows bytes of r/w internal flash when updating node software in a continuous manner. In most of continuous updating cases, the writing operation in internal flash is less, because only the replaced functions need to be written back to internal flash, but if free space is lacking (e.g., 9th and 20th updating cases), the r/w operations on internal flash must increase sharply.

To solve this storage fragmentation problem, we put forward a novel incremental reprogramming approach named RePage. This approach employs the paging mechanism to avoid internal flash space running out and reduce the r/w operations on flash by improving the utilization of cache.

3. Design and Implementation of RePage

In this section, we will discuss the paging mechanism based on function similarity degree and a novel function invocation method. The former mainly solves the storage fragmentation problem, while the latter reduces the r/w operation caused by caching and replacing page. At the same time, we also introduce a cache replacement algorithm that will improve cache hit rate.

3.1. Function Paging Mechanism Based on Function Similarity Degree. The reprogramming mainly adjusts some parameters or functionalities of program. Functions are still relatively independent, but the functionality adjustment tends to modify multiple relative functions. We define function similarity degree (FSD) to quantify such relationship among modified functions and have the function paging mechanism based on FSD. It will aggregate several similar functions into one function page and make each function page size-fixed.

By function paging, the program images will be equally divided into multiple function pages. Meanwhile, in order to ensure that each function page (fp_x) can be incremental and changeable flexibly, we add slop region (E_x) in each tail of each function page fp_x after consulting Koshy and Pandey's approach [12] shown in Figure 3. Slop region is specifically used for the location adjustment for a size-increased function.

Through a lot of reprogramming experiments, we found that the modification of function is usually unidirectional, which means that caller function will be changed with high possibility after any callee function changed; the opposite situation happens less. For example, if one function u's entry address has changed, all call instructions that call function u must be modified. It is clear that two functions have more common caller functions with more possibility being modified at the same time. Therefore, such unidirectional modification can describe the FSD between functions.

RePage figures the FSD, adopting the concept of collaborative filtering applied in recommendation system. Given functions u and v, make $N(u)$ and $N(v)$, respectively, be the callee function in u and v. The FSD between u and v can be described by Jaccard formula:

$$s_{uv} = \frac{|N(u) \cap N(v)|}{|N(u) \cup N(v)|}. \tag{2}$$

In fact, many functions do not have common caller function; namely, $|N(u) \cap N(v) = 0|$. Thus, we can firstly figure the callee function (u, v) of $|N(u) \cap N(v) \neq 0|$ for reducing the computation overhead.

After obtaining each S_{uv} between two functions, we calculate the whole function similarity degree (S) of function page with different weight as follows:

$$S(f_1, f_2, \ldots, f_i) = \sum_{u=1}^{i} \sum_{v=1}^{i} \beta_{uv} s_{uv}$$
$$\beta_{uv} = \frac{1}{\text{size}(u) * \text{size}(v)}, \tag{3}$$

where β_{uv} is weighted value, which is inversely proportional to the size of functions u and v. It ensures that if u and v own smaller size and have higher FSD, the whole degree of function page will be higher; otherwise, in the case of higher degree of functions u and v inasmuch as larger size, the similarity degree of current function page should be reduced.

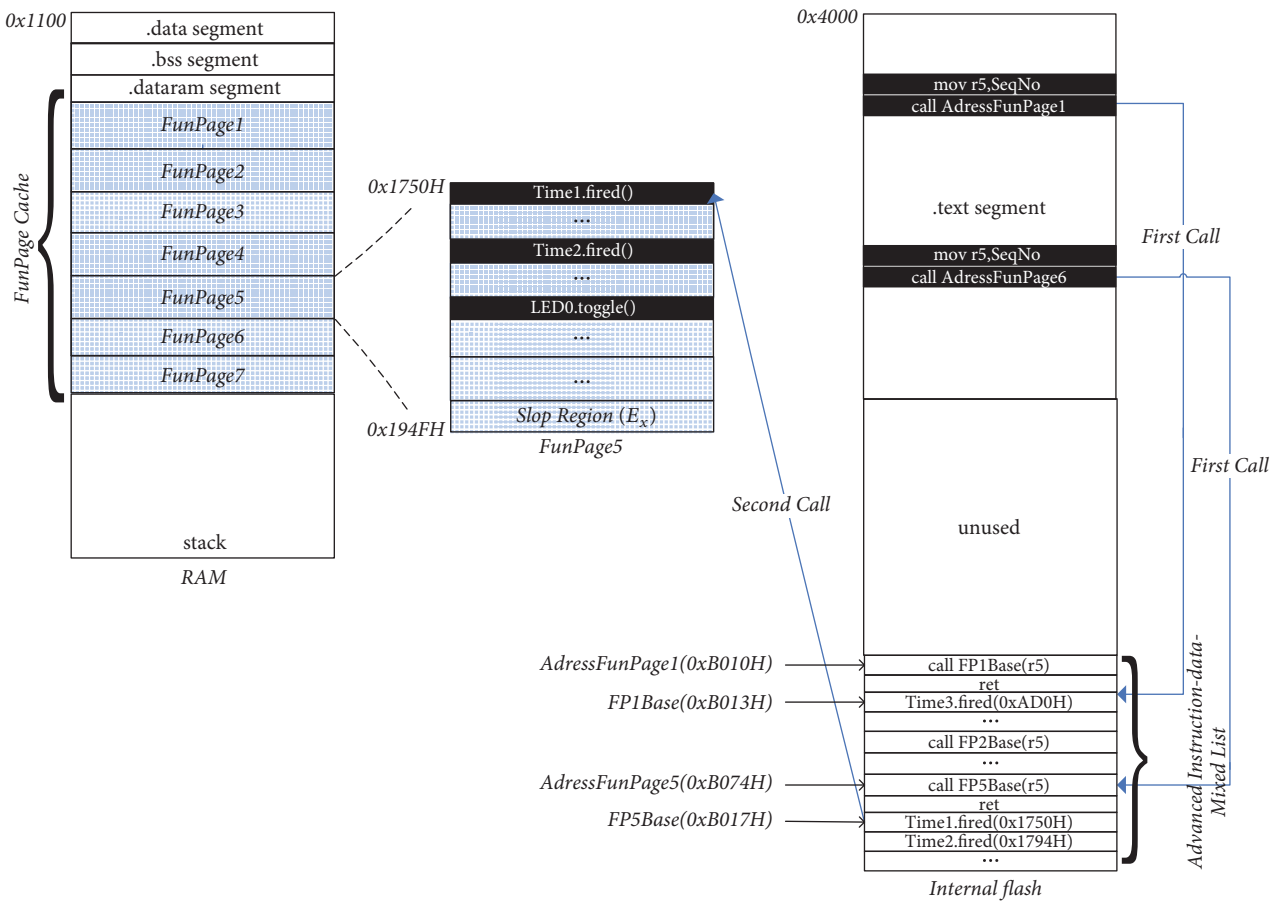

FIGURE 3: Memory layout and two-step calling process: for instance, function Time1.fired contained has address 0x1250 and is cached in page FunPage1. Once one call instruction needs to invocate Timer.fired, it will jump to AddressFunPagex saved in IDML and then obtain real entry address of Timer.fired (0xB013H) by register relative addressing. After obtaining real entry address, the invocation procedure is completed.

The aim of paging mechanism is to put more and more similar functions in one function page. So, we convert paging problem into an optimization problem shown in the following formula:

$$\begin{aligned} \max \quad & \sum_{k=1}^{K} S_k\left(f_i, f_{i+1}, \ldots, f_j\right) \quad i, j \in [1, F] \\ \text{s.t.} \quad & \mathrm{sz}\left(fp_x\right) = \mathrm{sz}\left(f_i\right) + \cdots + \mathrm{sz}\left(f_j\right) + \mathrm{sz}\left(E_x\right) \\ & \sum_{x=1}^{K} \mathrm{sz}\left(fp_x\right) < s_{\text{memory}}. \end{aligned} \tag{4}$$

The optimization objective is to make the similarity degree of function pages maximization. S_k is the similarity degree of function page k, which includes several functions. Parameter F is the number of functions within program image; K is the number of function pages which is much relative to page partition. $\mathrm{sz}(fp_x)$ is the size of each function page, which is up to the sum of sizes of similarity functions $(f_i, f_{i+1}, \ldots, f_j)$ and slop region (E_x). Due to slop region existing, it will make the size of paged program image bigger than the original one. Therefore, it needs to add the restricted condition to make the paged program image able to be accommodated by internal flash whose size is $\mathrm{sz}_{\text{memory}}$.

3.2. The Cache Replacement Algorithm of Function Page. The RAM is always rare, especially within MCU; thus, only small part of function pages can be cached, and most of function pages were still stored and executed in program. Therefore, a cache replacement algorithm is needed to choose which pages should be cached. Existing cache replacement algorithms such as least recently used (LRU) mainly focus on improving the execution efficiency, and RePage's algorithm focuses on reducing the r/w flash operations.

For example, a function page is replaced from cache, according to the LRU, due to not being modified recently. However, once this function page is modified, if a number of codes are changed, the page should stay in cache. Therefore, except the modification time, there is the other factor to decide whether page is being cached or not.

In terms of the hit rate of function page, we propose a recent modification range (RMR) algorithm that considers the modification times and range of each function page at the same time.

Table 2: Advanced instruction-data-mixed list.

Quantity of fun	Update times	AddressFunPageX		Function entry address		
		Call inst.	Ret inst.			
5	1	call FP1base(R5)	ret	fun1	fun2	⋯
10	3	call FP2Base(R5)	ret	fun12	fun23	⋯
7	8	call FP3Base(R5)	ret	Fun27	Fun9	⋯

With RMR, we define $\mathrm{RF}_{(i,m)}$, the replacement factor of function pages i after M times updating, as follows:

$$\mathrm{RF}_{(i,M)} = N_i^{(1-a)} * \left(\frac{\sum_{m=1}^{M} r_{(i,m)}}{M * \mathrm{sz}\left(fp_i\right)} \right)^a \quad a \in (0,1). \tag{5}$$

In formula (5), N_i represents modification times of function page i, and $r_{(i,m)}$ is modification range in page i and during mth updating. Rapidly increasing $r_{(i,m)}$ means that more codes changed within page i. Weight a is limited in range $[1, 0]$ and is used to adjust the impact of N_i and $r_{(i,m)}$ on $\mathrm{RF}_{(i,M)}$. In fact, if modification ranges tend to be the same in each updated function page, the replacement factor $\mathrm{RF}_{(i,M)}$ is largely up to modification times; otherwise, if modification ranges are enormously different with respect to different function pages, it is reasonable to increase impact of N_i, which will make large-modification-range pages be cached with high probability. Therefore, we define the weight a as relative to variance of $r_{(i,M)}$ as follows:

$$a = e^{-1/\sqrt{\sum_{i=1}^{n}(R - r_{(i,M)})^2/n}}. \tag{6}$$

In Mth updating, given n function pages that need to be updated, R is average of $r_{(i,M)}$ of n function pages that need to be updated. Clearly, if variance of $r_{(i,M)}$ is large, a tends to 1; otherwise, a tends to 0.

3.3. The Call and Load of Function. In our previous research EasiCache, function entry address must be changed because of being cached and replaced. If we modify every call instruction, it will cause a mass of r/w operations. To deal with that, we use a centralized management to save all entry address functions in a mixture call list and invocate the functions by two-step calling process.

In RePage, similar functions are put into one function page, and then all entry addresses must change when the function page is cached or replaced. For this, advanced instruction-data-mixed list (IDML) is given to manage the function address shown in Table 2.

Every item in the list focuses on one function page; the first two elements are the quantity of functions and the updating times of current function page. There is a call instruction applying register relative addressing and a return instruction within AddressFunPage$x$. The last column of item contains real function entry address. Once functions are put into certain function page, a sequence of functions in function page is fixed. And the real function entry address in IDML is saved in terms of the sequence.

When calling a function, two-step calling is needed. In the first calling, instruction pointer (IP) skips to the AddressFunPage rather than the real functions. Figure 3 shows the invocation procedure. Before entering the function, the sequence number (SeqNo) of the function in the function page should be assigned to a certain register, for instance, using register r5. Register r5 is taken as index register and used to continuously visit all functions in function page. In second calling, by register relative addressing, register r5 combining with base address (such as FP1base) points to the memory unit that stores the real entry address.

When a function page has been cached in the RAM or replaced for restore in internal flash, the function entry addresses must be changed. We can modify the entry address in the IDML to avoid modifying every instruction in functions. According to the difference between new address of function page and the original one, it is easy to count each new function entry address and write it in IDML.

Due to the register relative addressing, RePage needs to add a register assignment instruction before calling a function, shown in Figure 3. This adding process is transparent to user, but it may cause a register using conflict. We solved this problem by adding the instruction during the precompiling stage. We used C and assembly hybrid programming and directly put the instruction written by assembly-language in C-language source program. By this, the complier can automatically allocate register to avoid the register using conflict.

4. The Experimental Results and Analysis

To verify the function page mechanism, we need to test RePage performance in a continuous updating scenario. Continuously updating scenario is relative to the single updating scenarios, and it means using reprogramming approach to continuously update programs without redeploying node. Now most experiments of reprogramming approaches do not involve caching mechanism; thus, they usually took the single updating experiment for testing. Meanwhile, we mainly focus on continuous updating in experiment.

4.1. Experimental Scenario Introduction. The experiments take TinyOS operating system and TelosB node as the hardware and software platform comprehensively. TelosB nodes are equipped with CC2420, a 2.4 GHz ZigBee RF transceiver, and a 16-bit MCU MSP430F1611. With respect to continuous updating, we design two scenarios.

In the first updating set scenario, the experimental objects are standard routines program named Oscilloscope included

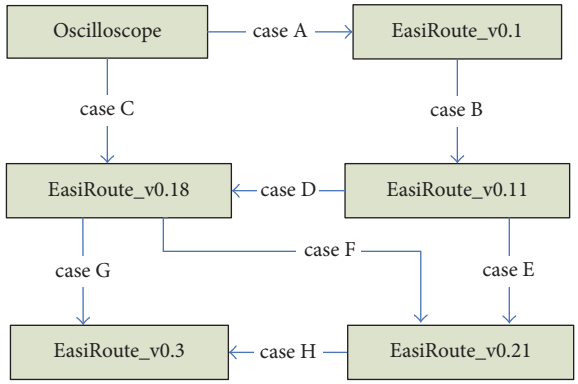

Figure 4: Roadmap of continuous updating.

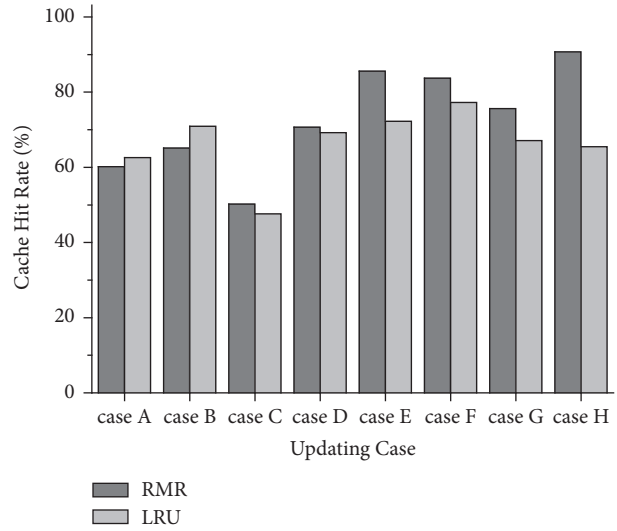

Figure 5: Cache hit rates of updating scenario according to alphabetical order.

in TinyOS application library and five versions of program named EasiRoute (versions from v0.1 to v0.3). These five programs are designed for EasiNet system that has been deployed in the Forbidden Palace Museum and applied for monitoring temperature and humidity [13]. Each version means a major upgrade. As shown in Figure 4, six programs are continuously updated according to alphabetical order.

Updating Case A. Oscilloscope is a standard data collection and transmission procedures. By adding routing functionality, we update it to EasiRoute_v0. 1.

Updating Case B. Update museum monitoring procedures from v0.1 to v0.11. After this update, a sensing-data storing functionality is added, and it realizes storing data in external flash chip.

Updating Case C. Directly update Oscilloscope to v0.11 and add routing and storing functionalities in resource-limited fog node.

Updating Case D. Update museum monitoring procedures from v0.11 to v0.18. After the update, a bug will be fixed. The bug leads that resource-limited fog node to send data too frequently to sink node, when museum is closed at night.

Updating Case E. Update v0.11 to v0.21. We add a load balancing algorithm and a sleep mechanism considering the battery energy remaining.

Updating Case F. Update v0.18 to v0.21. The new version will remove the sensing function from the node. It only retains routing function and forwards data from other ones.

Updating Case G. Update v0.18 to v0.3. After this update, node will install the UDP protocol that supports users directly interacting with the nodes through web browser.

Updating Case H. Update v0.21 to v0.3.

In the second updating set scenario, the above six programs will be randomly updated without fixed direction.

We will test RePage's performance from cache hit rate, energy cost, and storage cost and compare it with current reprogramming approaches such as EasiCache and Tiny Module-link.

4.2. Cache Hit Rate. First of all, we define cache hit rate as ratio between sum size of changed codes cached and the sum size of ones that need to be modified. Figure 5 shows hit rates of continuously updating cases according to alphabetical order shown in Figure 4. We set cache size in 6144 bytes, and the function page size is set in 512 bytes. As shown in Figure 5, we compare the RMR of RePage and Least Recently Used (LRU) algorithm.

RePage's adopted RMR algorithm considers modification times and range of each function page. However, in beginning phase of continuous updating, because the differences of modification range among each function page are small, weight a is close to 0, and impact of modification times gives more effect on replacement factor (RF). Performance of RMR is similar to LRU.

In updating case A, the codes of data acquisition and wireless transceiver have already been achieved in Oscilloscope. In addition, collection tree protocol (CTP) is usually taken as part of dead code and has been preinstalled by the dead code elimination technique referring to literature [7]. In fact, this updating procedure mainly activates CTP in application layer and adjusts the corresponding function according to EasiRoute_v0.1. However, only parts of changed codes are cached, and RMR's hit rate is only about 60% closed to LRU. In updating cases C and D, in order to fix the bug, some new functions need to be created in node and cannot be cached; thus, RMR's and LRU's hit rates are low.

During beginning phase, it is hard to predict which function pages are "hot." Therefore, RePage's algorithm has 62% hit rate compared with LRU's 63%. After beginning phase (from A to D), the average cache hit rate (from E to H) of RMR and LRU is improved. The reason is that RePage and LRU have enough history information to decide which pages should be cached. Especially in updating case H, two cached

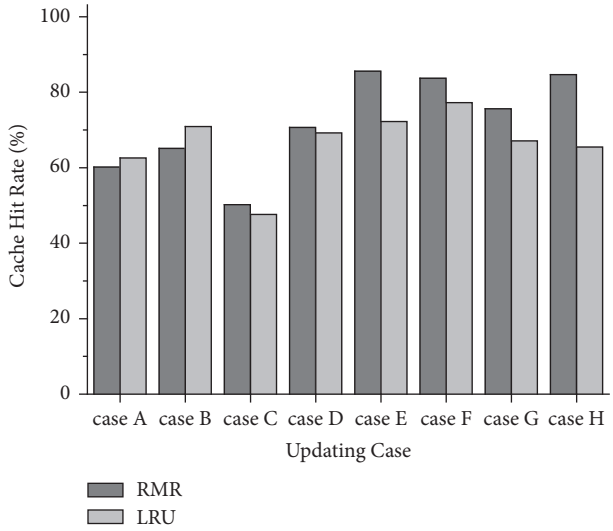

Figure 6: Cache hit rates of randomly updating scenario.

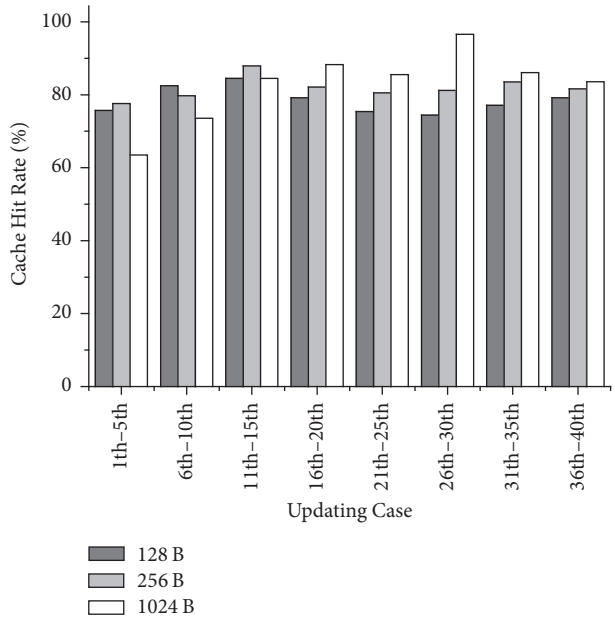

Figure 7: Cache hit rates of randomly updating scenario under different function page size.

function pages are replaced from cache during updating cases D and E according to LRU algorithm that only considers the modification times. But the two functions were largely modified in case B, so they have high RF and cannot be replaced according to RMR.

To further study the performance of RMR, we continuously update these six programs at random way without assigning the updating roadmap and observe hit rate of RMR and LRU. We record hit rates of five times updating and count the average value of hit rates taken as the average hit rate in random continuous updating. Figure 6 shows the average hit rates of 40 times updating. Obviously, in the first 15 updating cases, RMR's average hit rate reached 77.2% compared with 75.1% of LRU.

However, during 16th–40th updating, average hit rate of RMR increases and reaches 90.1%, and LRU only has 79.5%. The variation in performance between RMR and LRU is relative to modification range. In these updating cases, several function pages were not modified in most of cases and were therefore replaced, but once they are modified, this means a number of codes in these pages changed.

As shown in Figure 7, we also test the influence brought about by function page sizes to cache hit rate given cache capacity (6144 B). Small-sized function page means a flexible cache behavior. At the beginning phase, the 128 B function page obtained higher average hit rate. But limit of functions page size also makes some of larger functions be unable to be put in function page. Clearly, after 20 times continuous updating, the average hit rate of 128 B began to decline. But the larger-size function page (1024 B) still increases during 21st–35th updating. What needs to be pointed out is that the flexibility of 128 B page size makes updating hit rate more than 98% in the 10th and 14th updating cases, and its average hit rate has an advantage over 512 B and 1024 B page size during 1st–15th updating case. However, from the perspective of the stability of cache hit rate, the 512 B page size has a good performance during whole continuous updating procedure and is recommended.

4.3. Rebuilding Cost. The main energy cost of over-air reprogramming produced by communication cost is caused by wireless transmitting codes and the rebuilding cost is caused by r/w operations on memory. Table 1 has illustrated the energy cost of r/w operation on different memory. Obviously, energy cost in volatile memory such as RAM is far less than that in the nonvolatile memory such as internal/external flash. Take the TelosB nodes as example: the energy cost of r/w operation on the internal flash is 14.6 times that on RAM, and the energy cost of r/w operation on external flash is 19.3 times that on RAM. Therefore, when studying rebuilding cost of reprogramming, we focus on the number of r/w operations on internal/external flash (in/ex flash for short).

We compare RePage, Tiny Module-link [14], and EasiCache [10] under continuously updating scenario. Tiny Module-link firstly considered the energy cost of r/w operations. It rebuilds the function in low-power RAM, and then the updated function still needs to be written back into internal flash. EasiCache like RePage uses the low-power RAM caching frequently updating codes.

Tables 3 and 4 show the data size of r/w operations on in/ex flash of MCU in updating cases A–H with respect to three reprogramming approaches. RePage and EasiCache can directly modify codes in RAM and internal flash and do not require participation of external flash. In contrast, Tiny Module-link not only stores the incremental script in external flash but also must write back the modified functions into internal flash from RAM. So, Tiny Module-link must execute more r/w operations on internal flash than on external flash in some updating cases.

Table 3: Comparison of reading operations on external/internal flash among RePage, EasiCache, and Tiny Module-link (bytes).

		Case A	Case B	Case C	Case D	Case E	Case F	Case G	Case H
Tiny Module-link	R Ex flash	3278	1252	4072	1578	1314	872	4214	4562
	R In flash	3025	2456	2122	2894	2848	3010	35780	33678
EasiCache	R Ex flash	—	—	—	—	—	—	—	—
	R In flash	902	614	1322	918	594	828	36678	32172
RePage	R Ex flash	—	—	—	—	—	—	—	—
	R In flash	356	314	422	204	768	266	2232	1866

Table 4: Comparison of writing operations on external/internal flash among RePage, EasiCache, and Tiny Module-link (bytes).

		Case A	Case B	Case C	Case D	Case E	Case F	Case G	Case H
Tiny Module-link	W Ex flash	3678	1252	4072	1578	1314	872	4214	4562
	W In flash	3214	2022	3678	2760	3090	1320	30258	31142
EasiCache	W Ex flash	—	—	—	—	—	—	—	—
	W In flash	578	456	624	1342	1090	1278	30958	31588
RePage	W Ex flash	—	—	—	—	—	—	—	—
	W In flash	636	278	728	186	1488	672	2356	2712

EasiCache and RePage put codes in low-power RAM and effectively reduce the r/w operations on internal flash. The average hit rate of RePage is 6.1% higher than EasiCache in updating B, D, and F of RePage. The reason is that paging mechanism puts the similar functions clustering in one function page and keeps the page in cache. In contrast, EasiCache does not consider the relationship between functions, and sometimes its behavior is like a bouncing ball, which means several functions are replaced and cached repeatedly. High hit rate means that more functions have been saved in the RAM; and there is no need to read internal flash, and written back codes are less.

Updating case E is special. In this scenario, although the hit rate of RePage is still higher than EasiCache, one function page needs to be replaced and written back to the internal flash. Exchanging 512 B function page brings about 34% increase of r/w operations.

In updating cases G and H, several functions are inserted in program due to adding UDP protocol. They need to be written into the internal flash. EasiCache and Tiny Module-link E adopt the same strategy to preserve inserted functions in the free space at the end of the main program (.text segment). However, in this case, the size of function, writing in the internal flash, has been beyond the capacity of internal flash. Given both approaches, runtime system has to readjust the storage location for all functions in internal flash and increase a large number of r/w operations on internal flash. RePage uses paging mechanism and makes each page own independent slop region, which ensures the adjustment occurring in function page. Compared with EasiCache and Tiny Module-link, the number of r/w operations on in/ex flash of RePage decreases by 93.8% and 93.0%.

4.4. Storage Cost. Caching codes inevitably causes a fragmentation problem. Figure 8 shows the fragmentation degree in continuous upgrading cases with respect to RePage and EasiCache. The definition of fragmentation degree has been given by formula (1).

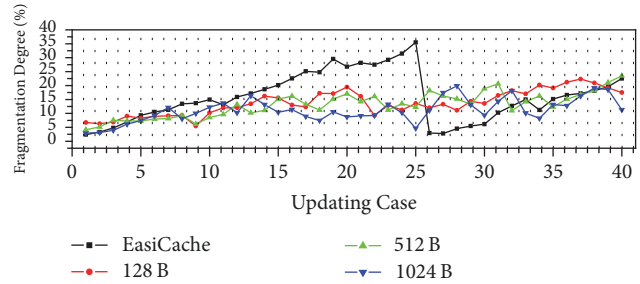

Figure 8: Fragmentation degrees of randomly continuously updating scenario.

In RePage, each function page has an independent slop region $E(x)$ that actually causes storage fragmentation. Therefore, in the beginning of the continuous updating cases, the fragmentation degree of both EasiCache and RePage is not zero. However, due to uncertain function size, after several times of upgrading, the fragmentation degree of EasiCache sharply increases.

EasiCache can reduce fragmentation degree by adjusting function position. For example, during the 5th case, some instructions are deleted from several modified functions which become small, and runtime system switches position of both size-decreasing functions and size-increasing functions. However, the fragmentation degree is not stable. With increased modified functions, after 13th updating case, the size-decreasing functions are unable to provide enough space for size-increasing functions which must be written back to internal flash. Clearly, fragmentation degree of EasiCache is increased generally. In the 25th upgrading case, when the fragmentation degree exceeds 35%, it is hard to cache

Table 5: The decreasing of execution efficiency (%).

Program name	Dec.	Program name	Dec.
Oscilloscope	10.6	EasiRoute_v0.1	12.3
EasiRoute_v0.11	14.5	EasiRoute_v0.18	13.6
EasiRoute_v0.21	12.6	EasiRoute_v0.3	17.9

new functions. All functions have to be readjusted in 26th updating case and fragmentation degree drops to about 3%.

With respect to RePage, each function page has own slop region. The function location adjustment only impacts these functions that are clustered in the same function page. In addition, the size of function pages will also have influence on the fragmentation degree. In Figure 8, we compared fragmentation degree of all upgrades with different page sizes (128 B, 512 B, and 1024 B). Large-size function page contains more similar functions than small-size one and its slop region does not obviously increase. Therefore, 1024 B page has fewer fragmentation spaces and high utilization in the beginning. However, in large-size page, the slop region is relatively far from each function, and the adjustment process is also more difficult than small-size page. With the progress of continuous updating, for the large-size page, each adjustment procedure will cause fragmentation degree to have serious fluctuation obviously shown in vibration curve in Figure 8.

With respect to small-size page, it is convenient to adjust the function entry address due to slop region being closed to functions. Therefore, the average of fragmentation degree of 128 B and 512 B page function decreases by 5.5% and 10.8% compared to 1024 B. Figure 8 shows that the curves of fragmentation degree of 128 B and 512 B also appear smoother than 1024 B.

4.5. The Impact of Two-Step Calling Process on Execution Efficiency. In RePage, invocating functions need two-step calling process. In the first step, it is necessary to assign register (r5) before calling function. In the second step, RePage employs register relative addressing to find the real entry address. With respect to two-step calling process, add an assignment (mov), a calling (call), and a return instruction (ret.). The three instructions, respectively, need 3, 8, and 5 clock cycles. Therefore, a total of 16 clock cycles are added, caused by two-step calling process.

Table 5 shows the decrease of execution efficiency after two-step calling, which includes six programs. The biggest impact is EasiRoute_v0.3, where the execution efficiency is down by 17.9% from the original program. This program is more complex due to joining the UDP protocol module. The execution efficiency of the other five programs declines by 10.6% at least. Obviously, the two-step calling process does have influence on the execution efficiency. However, considering the dynamic updating of the fog computing system, two-step calling is able to reduce modification and manages the function page facilitating. Compared with the energy cost of reprogramming, the added energy cost is negligible, caused by decreasing of execution efficiency.

5. Related Work

The energy efficiency of reprogramming has been a key issue in resource-limited networks always taken as front-end network of fog computing. The early reprogramming researchers pay less attention to energy efficiency and burden of network. For example, Deluge [15] needed to transmit the complete program image and related updating protocol. During the transmission period, reprogramming process makes network unavailable in a long time. Stream [16] attempts to reduce the size of data transmitted by preinstalling the reprogramming protocol in the target node, but it still transmits the entire program image. CORD [17] employs two stages to reduce energy cost. In the first stage, new image is given to special nodes, and then, in the second stage, these special nodes will transmit new image to the rest of ones. Mobile Deluge [18] introduces a mobile base station to ensure the receiver and transmitter within one hop. By this way, it improves quality of transmission link and decreases burden of network. In addition, according to discrete and dynamic characters of resource-limited front-end networks, researchers have proposed the encoding reprogramming approach [19–21]. They adopt the redundant code to guarantee success of transmission. All of them need to transmit the whole image and even redundant codes that caused huge energy cost.

In order to reduce the communication cost in reprogramming, some studies introduce compression algorithm, such as adaptive compression algorithm [22]. Tsiftes et al. directly employ GZIP to compress program image [23]. However, compressing image means a decompression procedure needs to be completed in local resource-limited fog nodes. This procedure must bring up computation cost and storage cost.

In recent years, a lot of incremental reprogramming approaches [1–8] were proposed. The concept of incremental reprogramming is to minimize the communication overhead by merely transmitting the different binary codes between the new and old images. Current incremental reprogramming approaches can fall into two categories: one increases the similarity between new and old images and the other improves the algorithm's efficiency of generating different codes.

In the works of Zephyr [2] and Hermes [1], researchers attempt to fix entry address of function and static variable to improve the similarity of images. Li et al. [3] designed an update-conscious compiler that rearranges registers in new program image according to old version, and, by this way, the similarity degree between two versions is largely improved. On the other side, Hu et al. [5] propose reprogramming with minimal transferred data (RMTD) algorithm that is based on Rsync algorithm; this algorithm reduces the time complexity of Rsync from $O(n^3)$ to $O(n^2)$, but the space complexity reaches $O(n^3)$. That may bring about heavy storage burden for upper computer given large-size program image. Dong et al. combined the advantages of Zephyr and RMTD and proposed R2 [7] and R3 [8], and both approaches take advantage of relative address codes to initialize reference addresses for improving similarity. In addition, they also modify the RMTD algorithm to reduce the space complexity to $O(n)$. All above incremental reprogramming approaches need to rebuild new program image locally. Meanwhile,

rebuilding procedure may incur a large number of read and write operations on nonvolatile memory with high energy cost.

Most of current reprogramming approaches are designed based on the traditional resource-limited networks such as sensor network; the program running in node is hard to change, and researchers consider less rebuilding process within frequent and continuous updating. In Zephyr's work [3], for example, researchers have tested rebuild cost of r/w operation on nonvolatile memory and questioned current reprogramming approach. Kim et al. [14] put forward Tiny Module-link that rebuilds the new program image in the RAM. But it still requires writing back new function into nonvolatile memory. In addition, Koshy and Pandey [12] design extra slop region at the end of each function and support directly modifying function in the nonvolatile memory. In other studies, even if low-power volatile memory is used, the purpose is not absolutely to reduce the rebuilding cost. For example, in Elon's work [24], the researcher uses RAM to store frequently updated code and solves the writing flash failure problem caused by the battery voltage drop.

6. Conclusion

Fog computing system requires frequent task rearrangement, and thus fog nodes may experience continuous update by over-air reprogramming. The energy cost is always bottleneck of reprogramming applied in fog computing system. This paper introduces an incremental reprogramming approach based on function page mechanism applied in fog computing, named RePage. With respect to increment reprograming approaches, they must rebuild new program image in local node, which may incur an amount of rebuilding energy cost. For decreasing the rebuilding cost, RePage tries to cache frequently updated function in low-power volatile memory and use function paging mechanism to combine similar functions into one function page for mitigating fragmentation degree.

In order to improve hit rate, we design a novel page replacement algorithm named recent modification range (RMR) that considers the impact factor of modification range and times at the same time.

Through the continuous update experiment, we showed the performance of RePage from the cache hit rate, storage cost, and energy cost. In future work, we will continue to research differences-code generation method according to the function paging mechanism. This method will simplify rebuilding instructions according to such function similarity and further reduce the amount of transmitting data.

Conflicts of Interest

The authors declare that there are no conflicts of interest.

Acknowledgments

This research work is supported by National Natural Science Foundation of China (NSF) (Grant no. 61502427), by the Zhejiang Provincial Natural Science Foundation of China (Grant no. LY16F020034), and by Key Research and Development Project of Zhejiang Province (no. 2015C01034 and no. 2015C01029).

References

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

[2] 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.

[3] W. Li, Y. Zhang, J. Yang, and J. Zheng, "Towards update-conscious compilation for energy-efficient code dissemination in WSNs," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 6, no. 4, p. 14, 2009.

[4] J. Jeong and D. Culler, "Incremental network programming for wireless sensors," in *Proceedings of the First Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks, IEEE SECON '04*, pp. 25–33, Santa Clara, CA, USA, 2004.

[5] 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 IEEE 6th International Conference on Mobile Adhoc and Sensor Systems, MASS '09*, pp. 160–167, China, October 2009.

[6] M. Ajtai, R. Burns, R. Fagin, D. D. Long, and L. Stockmeyer, "Compactly encoding unstructured inputs with differential compression," *Journal of the ACM*, vol. 49, no. 3, pp. 318–367, 2002.

[7] W. Dong, Y. Liu, C. Chen, J. Bu, C. Huang, and Z. Zhao, "R2: incremental reprogramming using relocatable code in networked embedded systems," *Institute of Electrical and Electronics Engineers. Transactions on Computers*, vol. 62, no. 9, pp. 1837–1849, 2013.

[8] W. Dong, B. Mo, C. Huang, Y. Liu, and C. Chen, "R3: Optimizing relocatable code for efficient reprogramming in networked embedded systems," in *Proceedings of the 32nd IEEE Conference on Computer Communications, IEEE INFOCOM '13*, pp. 315–319, Italy, April 2013.

[9] J. Polastre, R. Szewczyk, and D. Culler, "Telos: enabling ultra-low power wireless research," in *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks (IPSN '05)*, pp. 364–369, April 2005.

[10] J.-F. Qiu, D. Li, H.-L. Shi, W.-Z. Du, and L. Cui, "EasiCache: A low-overhead sensor network reprogramming approach based on cache mechanism," *Jisuanji Xuebao/Chinese Journal of Computers*, vol. 35, no. 3, pp. 555–567, 2012.

[11] "MSP430F15x, MSP430F16x, MSP430F161x Mixed Signal Microcontroller," http://focus.ti.com/lit/ds/symlink/msp430f1611.pdf.

[12] 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, Turkey, February 2005.

[13] 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 2010*, USA, December 2010.

[14] 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.

[15] 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, ACM, 2004.

[16] 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 (IEEE INFOCOM '07)*, pp. 928–936, USA, May 2007.

[17] L. Huang and S. Setia, "CORD: Energy-Efficient Reliable Bulk Data Dissemination in Sensor Networks," in *Proceedings of the IEEE Conference on Computer Communications (INFOCOM '08)*, pp. 574–582, Phoenix, AZ, USA, 2008.

[18] X. Zhong, M. Navarro, G. Villalba, X. Liang, and Y. Liang, "MobileDeluge: Mobile code dissemination for wireless sensor networks," in *Proceedings of the 11th IEEE International Conference on Mobile Ad Hoc and Sensor Systems, MASS 2014*, pp. 363–370, USA, October 2014.

[19] A. Hagedorn and D. Starobinski, "Rateless deluge: over-the-air programming of wireless sensor networks," in *Proceedings of the IEEE International Conference on Information Processing in Sensor Networks (IPSN '08)*, Proceeding of ACM, April 2008.

[20] M. Rossi, G. Zancas, L. Stabellini, R. Crepaldi, A. F. Harris III, and M. Zorzi, "SYNAPSE: A network reprogramming protocol for wireless sensor networks using fountain codes," in *Proceedings of the 5th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks, SECON 2008*, pp. 188–196, USA, June 2008.

[21] W. Du, Z. Li, J. C. Liando, and M. Li, "From rateless to distanceless: enabling sparse sensor network deployment in large areas," in *Proceedings of the 12th ACM Conference on Embedded Networked Sensor Systems (SenSys '14)*, pp. 134–147, Memphis, Tenn, USA, November 2014.

[22] C. M. Sadler and M. Martonosi, "Data compression algorithms for energy-constrained devices in delay tolerant networks," in *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems (SenSys '06)*, pp. 265–278, ACM, 2006.

[23] N. Tsiftes, A. Dunkels, and T. Voigt, "Efficient sensor network reprogramming through compression of executable modules," in *Proceedings of the 5th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON '08)*, pp. 359–367, June 2008.

[24] W. Dong, Y. Liu, C. Chen, L. Gu, and X. Wu, "Elon: Enabling efficient and long-term reprogramming for wireless sensor networks," *ACM Transactions on Embedded Computing Systems*, vol. 13, no. 4, article no. 77, 2014.