Remote Progressive Update for Code-Paging Firmware in Flash-Based Networked Embedded Systems

Jinsik Kim[1]
[1]University of California, Irvine
CA, USA 92697-2625
jinsikk@uci.edu

Pai H. Chou[1,2]
[2]National Tsing Hua University
Hsinchu, Taiwan 30013
phchou@uci.edu

ABSTRACT

Firmware update over a network connection is an essential but expensive feature for many embedded systems due to the relatively high power consumption and limited bandwidth. This work proposes a page-level, link-time technique that minimizes not only the size of patching scripts but also perturbation to the firmware memory, over the entire sequence of updates in the system's lifetime. Experimental results show our technique to reduce the energy consumption of firmware update by 38–42% over the state-of-the-art.

Categories and Subject Descriptors

D.3.4 [**Processors**]: Code generation

General Terms

Algorithms, Management, Measurement, Performance

Keywords

High-level analysis, NOR Flash memory, Page, Diff, Clycomatic complexity, Progressive code update, Embedded systems

1. INTRODUCTION

The ability to update firmware over a network link is becoming an increasingly important feature. Updates are applied for enhanced security, feature upgrade, bug fixes, and conformance to newly finalized industry standards, among many reasons. Firmware is usually stored in nonvolatile memory such as EEPROM or Flash. Remote firmware update can be an expensive process for many embedded systems. For instance, a wireless sensor node that is deployed remotely or deeply embedded may need to run on battery or harvested power, and RF transceivers and flash memory access almost always consume higher power than any other component in the system by at least an order of magnitude. While one may overwrite the entire firmware image, it is less desirable due to unnecessary wear-and-tear and potentially long time. The problem is exacerbated if the firmware update process is done by peers.

Previous works have attempted to reduce the cost of firmware update by transmitting differences in the code images. Even if the difference is small, any change in code size can cause shift in potentially unchanged data, translating into more energy consumption, delay, and additional wear-and-tear of the flash memory. Although researchers have proposed leaving gaps to avoid anticipated shifts, their effectiveness over the *entire lifetime* of the system has not been demonstrated.

We propose a new technique, called Remote Progressive Firmware Update (RPFU), which improves over the state-of-the-art considering the characteristics of different functions in not only grouping them in the same pages but also ordering them within the page. This step is performed during linking after compilation. The resulting code image translates into a small diff script to minimize energy for transmission. Moreover, the diff script performs minimal shifting, thereby reducing the number of unnecessary rewrites to the flash memory. A distinguishing technique is that our technique *evolves* well over the entire lifetime of the system, not just between some randomly chosen pair of successive versions. We show the effectiveness over at least nine consecutive versions of real applications.

2. RELATED WORK

Previous works have studied the cost reduction of firmware update. The costs are associated with the communication and the number of rewrites. Note that low communication cost does not automatically imply fewer rewrites, because one may transmit a small script that commands many data movements.

To reduce communication cost, previous works have considered transmitting the difference of code between different versions [1, 2, 3, 4]. They have the effect of reducing communication cost but unfortunately do not consider the flash memory characteristics. The main difference with flash memory is that data modification requires explicit erasure before writing, as it cannot simply overwrite existing data. Moreover, erasure is done in units of pages. Erasure costs power, time, and wear-and-tear. Conventional memory management techniques, when applied to flash memory, have the problem of shifting of unchanged data in order to accommodate newly written data of a different size. To address this problem, fragmented layout [5] has been proposed by inserting gaps between erasure units. However, this leads to memory fragmentation, and their effectiveness over a series of firmware updates has not been demonstrated.

Another problem with shifting code is control-flow dependency. That is, if a callee is moved, then all callers of that function must be updated with the new address, and these callers may reside on several different pages. A common solution is to make an indirect call through a jump table, so that only the jump table needs to be updated, but this incurs runtime overhead each time. To minimize the domino effect of code shift, feedback linking [6] takes a

ISLPED'09, August 19–21, 2009, San Francisco, California, USA.

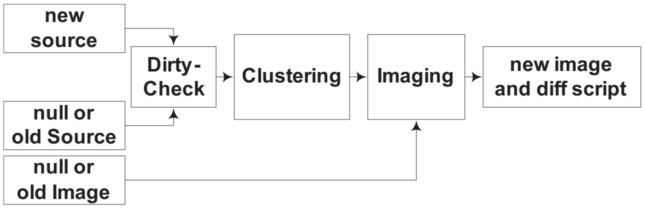


Figure 1: Framework Block Diagram

```
PROGRESSIVEUPDATE($NSC, OSC, OPBI$)
$MF \leftarrow$ DIRTYCHECK($NSC, OSC$)
$URPBI \leftarrow$ CLUSTERING($MF, OPBI$)
$DS, PBI \leftarrow$ IMAGING($MF, URPBI$)
return $DS, PBI$
```

Figure 2: Top-level algorithm.

code-layout approach by placing modified functions at the end of an image or gaps between functions. However, it does not analyze the callers to effectively minimize their updates when the callee is shifted.

Our proposed work makes several contributions. It computes a code layout based on the structure of the program, so that it will be efficient to update throughout the system's entire lifetime. That is, it minimize not only the difference between two arbitrary successive versions, but also the total Hamming distance from the first to the last version. This means it will be not only energy efficient to transfer, since the *diff* script is small, but also energy efficient to patch, since the shifting and rewriting are minimized.

3. OVERALL FRAMEWORK

The top-level algorithm PROGRESSIVEUPDATE() shown in Fig. 2 calls two procedures named CLUSTERING and IMAGING for the purpose of generating code images. The symbols in all pseuduocode in this paper are listed in the Appendix. The images are organized into pages that match the native page size of the flash memory. The CLUSTERING procedure performs grouping of functions into pages, while the IMAGING procedure inputs these groups and produces the final layout as well as a *diff* script. The diff script contains commands and difference data for updating the firmware, and it is what is actually disseminated to the sensor nodes over the communication link. In Fig. 1, CLUSTERING and IMAGING procedures are performed on the host side, while the diff script is parsed on the deployed node side.

Without loss of generality, for the purpose of our experiments, we assume power characteristics of Eco platform [7] and NOR flash memory [8]. The characteristics are in Tables 1 and 2, respectively. We also make several other assumptions. First, this method applies to updates of monolithic binaries such as operating systems, virtual machine engines, and scripting engines as well as monolithic binaries on real-time systems without memory management. The updates are through a communication interface that is costly to operate, by consuming relatively high power (e.g., RF module of a wireless sensor node) or is relatively slow. We also assume NOR type of flash memory for firmware due to its ability to perform byte-reading and page-erasing, and it is the most popular form of nonvolatile program memory for embedded systems.

Table 1: Eco power characteristics

Parameter	current	Parameter	current
RF RX	10.5 mA	CPU Active	3mA
RF TX	19 mA	CPU Powerdown	2 μA
EEPROM	5 mA	ADC	0.9 mA

Table 2: Flash energy consumption (unit: μJ/byte)

Component	Read	Write	Erase
AT29C010A	0.25	0.48	0.48

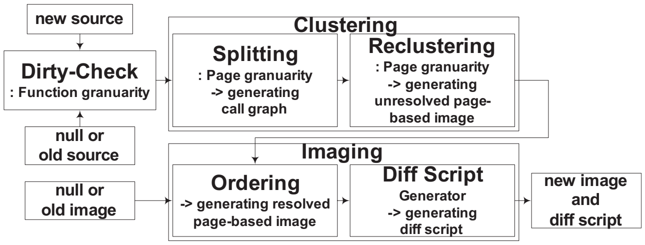


Figure 3: Framework in detail

4. CLUSTERING WITHIN PAGES

The CLUSTERING procedure performs grouping of functions to fit in pages whenever possible, such that the number of references across pages (i.e., caller to callee) is minimized. For code updates based on flash memory, modifying a function may occur in two different ways: modification in-place and page reassignment. In the first case, modifying a function in-place may mean shifting code of all the other functions that are placed after the modified function within the same page. This will in turn cause other pages containing references to those shifted functions to be updated as well, and this represents the worst case of modification. In the second case, all pages containing references to the modified function need be modified. This procedure may need to modify some of the pages containing references to the modified function.

The CLUSTERING procedure is further divided into SPLITTING and RE-CLUSTERING. SPLITTING extracts the call graph structure for the functions. If this is the very first version of the program, then the graph covers the entire set of functions. Otherwise, it covers the set of modified functions plus those in the existing page-based image. The call graph structure is fed to the next step, RE-CLUSTERING. The purpose of RE-CLUSTERING is to create either a good initial grouping or minimally different grouping that will result in low energy consumption when transmitted or updated. Each group of functions will fit within a page and then ordered to further minimize intra-page shifts. The CLUSTERING algorithm is shown as a flow chart and pseudocode in Figs. 3 and 4. The SPLITTING and RE-CLUSTERING procedures are presented next.

4.1 Splitting

SPLITTING inputs a list of modified functions and the page-based image from the previous version. It first calls CGDATASTRUCTURE to analyze the caller-callee relationship and the complexity of the functions, and then partitions them among the pages. Each function is assumed to fit within a page. Then, SPLITTING calls PBCALLGRAPH to construct a *page-based call graph* (PBCG), where the vertices represent the functions and the edges represent

```
CLUSTERING($MF, OPBI$)
    ▷ See Appendix for list of symbols
$DSS, PBCG \leftarrow$ SPLITTING($MF, OPBI$)
$URPBI \leftarrow$ RECLUSTERING($DSS, PBCG$)
return $URPBI$
```

Figure 4: CLUSTERING **algorithm.**

```
SPLITTING($MF, OPBI$)
$DSS \leftarrow$ CGDATASTRUCTURE($MF$)
$PBCG \leftarrow$ PBCALLGRAPH($DSS, OPBI$)
return $DSS, PBCG$

PBCALLGRAPH($DSS, OPBI$)
$PBCG \leftarrow \{\}$
while $DSS \neq \{\}$
    do $f_k \in DSS$
        if $f_k =$ ENLARGEDFUNCTION
            then $PBCG \leftarrow$ ENFCOST($f_k, OPBI, PBCG$)
            else if $f_k =$ SHRUNKFUNCTION
                then $PBCG \leftarrow$ SHFCOST($f_k, OPBI, PBCG$)
            else
                $PBCG \leftarrow$ RMFCOST($f_k, OPBI, PBCG$)
return $PBCG$
```

Figure 5: SPLITTING **and** PBCALLGRAPH **pseudocode.**

caller-callee relationships, and the vertices are grouped by pages. The objective of SPLITTING is to minimize the cost of the PBCG.

The cost of update is directly related to (1) the number of pages that need to be updated and (2) the style of update for each function. A page needs to be updated if it contains either a modified function or references to a relocated function. Note that a modified function may be the same size, enlarged, shrunk, removed, or newly added with respect to the previous version. The update style for each function can be further classified into (1) *in-place* update, i.e., same starting address on the same page; (2) *anew-in-place* update, i.e., same starting address on the same page with *in-place* update; (3) writing the modified function to free space, or *hole*, in another page; (4) *shifting* some other functions' code on the same page in addition to writing the modified function; (5) *anew-shifting* the some other functions as *shifting* that on the same page in addition to writing the modified function; (6) allocation of a *new* page; (7) *removing* a page. The energy for these update styles are modeled as follows.

$$\begin{aligned} E_{\text{inplace}}(\Delta(f_k)) = {} & (E_{\text{cpu}}(FLASH+BUF)+E_{\text{buf}}(read+write) \\ & +E_{\text{flash}}(read+erase+program))\times size(PAGES(f_k)) \\ & +(E_{\text{rf}}+E_{\text{cpu}}(RF))\times(size(R_{\text{shift}}(f_k))+size(\Delta(f_k))) \end{aligned} \tag{1}$$

$$E_{\text{anewinplace}}(\Delta(f_k)) = (E_{\text{cpu}}(FLASH+BUF)+E_{\text{buf}}(read+write) \\ +(E_{\text{rf}}+E_{\text{cpu}}(RF))\times(size(R_{\text{shift}}(f_k)+\Delta(f_k))) \tag{2}$$

$$\begin{aligned} E_{\text{hole}}(\Delta(f_k)) = {} & (E_{\text{cpu}}(FLASH+BUF)+E_{\text{buf}}(read+write) \\ & +E_{\text{flash}}(read+erase+program))\times size(PAGES(f_k)) \\ & +(E_{\text{rf}}+E_{\text{cpu}}(RF))\times(size(R_{\text{shift}}(f_k))+size(\Delta(f_k))) \\ & +E_{\text{flash}}(erase+read+program)\times size(PAGES(R(f_k))) \end{aligned} \tag{3}$$

$$\begin{aligned} E_{\text{shift}}(\Delta(f_k)) = {} & ((E_{\text{cpu}}(FLASH+BUF)+E_{\text{buf}}(read+write) \\ & +E_{\text{flash}}(read+program))\times(size(PAGES(I(f_k))) \\ & +(E_{\text{rf}}+E_{\text{cpu}}(RF))\times(size(R(f_k))+size(\Delta(f_k))) \\ & +E_{\text{flash}}(erase+read+program)\times size(PAGES(R(f_k))) \end{aligned} \tag{4}$$

$$E_{\text{anewshift}}(\Delta(f_k)) = (E_{\text{rf}}+E_{\text{cpu}}(RF))\times(size(R(f_k))+size(\Delta(f_k))) \tag{5}$$

$$\begin{aligned} E_{\text{new}}(f_k) = {} & (E_{\text{rf}}+E_{\text{cpu}}(RF))\times(size(R_{\text{shift}}(f_k))+size(\Delta(f_k))) \\ & +E_{\text{flash}}(erase+read+program)\times size(PAGES(R(f_k))) \end{aligned} \tag{6}$$

$$E_{\text{remove}}(f_k) = E_{\text{flash}}(erase+read+program)\times size(PAGES(R(f_k))) \tag{7}$$

$E_{\text{cpu}}(FLASH)$, $E_{\text{cpu}}(BUF)$, and $E_{\text{cpu}}(RF)$ represent the energy consumption of CPU execution for flash memory, a buffer, or RF communication, respectively. $E_{\text{buf}}()$, $E_{\text{flash}}()$, and $E_{\text{rf}}()$ represent the energy consumption of a buffer execution, flash memory execution, and RF transmission, respectively. $R(f_k)$ represents references to f_k, and $R_{\text{shift}}(f_k)$ represents references to functions shifted by f_k. $I(f_k)$ represents functions shifted by f_k. $PAGES()$ represents the page(s) at which input function(s) are located.

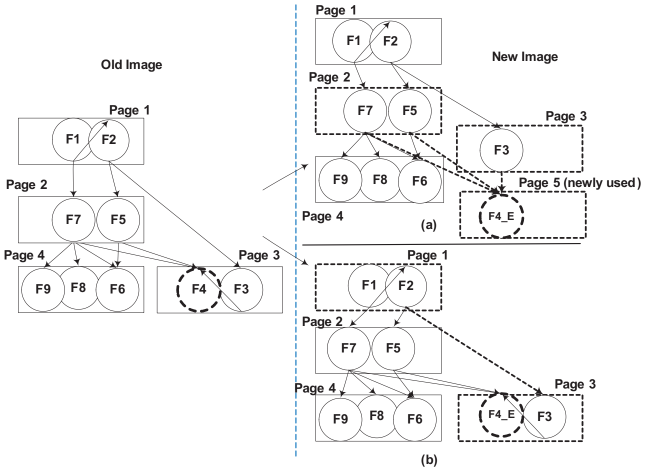


Figure 6: Splitting in case of an enlarged function, F_4. (a) the enlarged function, $F4\_E$, moved to the page # 5 : page 1, page 3, and page 5 to be updated. (b) $F4\_E$ in place : page 1 and page 3 to be updated.

In Fig. 6, F_4 is enlarged and renamed F_4_E. One way is to write F_4_E to a newly allocated Page 5, which necessitates updates to F_4's callers on Page 2. Another way is to write F_4_E back to Page 3 by shifting F_3; although this does not affect the callers of F_4, it affect the callers of F_3 and thus requires update to Page 1.

4.2 Reclustering

RECLUSTERING adds the modified functions (MF) to the page-based call graph ($PBCG$) to generate an unresolved page-based image. Its objective is to minimize the number of inter-page references. To do this, RECLUSTERING explores grouping functions that are related as *parents* (callers) of a common *child* (callee), (b) *cousins*, i.e., nodes with common callees, and (c) parent and its *children* (callees) or subset thereof. Fig. 7 shows these three ways to cluster with respect to the function F_7. The objective of reclustering is to minimize the number of inter-page references. Formally,

$$\text{minimize} \sum_{k=1}^{n}(NR^{Page_{all}}(Page_k)) \tag{8}$$

where $NR =$ number of inter-page references, $n =$ number of pages, $k =$ page number, and $Page =$ erasure unit of flash memory. The expression $NR^{callee}(callers)$ counts the number of references in $callers$ to $callee$, and $NR^{Page_{all}}(Page_k)$ means the number of references in all pages to $Page_k$. For example, in Fig. 6, $NR^{f_6}(f_7) = 1$ and $NR^{f_6}(f_7, f_5) = 2$. Equations (9), (10), and (11) express the number of reference crossing pages ($NRCP$) after clustering with one of a parent node, a cousin node, and a child node. RECURSIVECLUSTERING finds and merges a function into its parent, cousin, or child page.

$$NRCP_{parent} = \sum_{k=1}^{n}(NR^{Page_{all}}(Page_k)) - NR^{f_j}(f_i) \tag{9}$$

$$NRCP_{cousin} = \sum_{k=1}^{n}(NR^{Page_{all}}(Page_k)) - NR^{f_i}(Child(f_i, f_j)) \tag{10}$$

$$NRCP_{child} = \sum_{k=1}^{n}(NR^{Page_{all}}(Page_k)) - NR^{f_i}(f_j) \tag{11}$$

In addition, FINDPTRNODE finds the node that will be clustered with N based on Equation (12) comparing with $PreSv$ and $CurSv$. The found node is called as a parter node, $PtrN$ which is used as an input for CLUSTERINGTWONODES. CLUSTERINGTWONODES

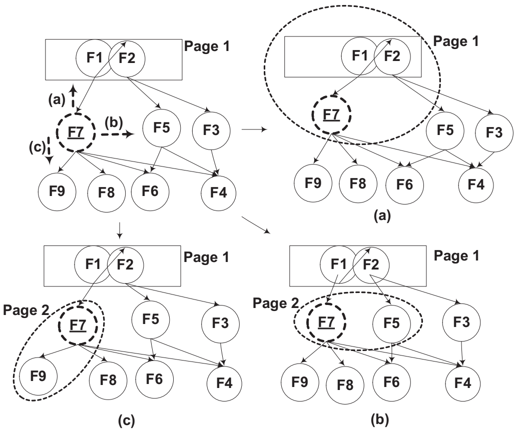


Figure 7: Different Clustering Ways

clusters N with $PtrN$ and adds the clustered node to an unresolved page-based call graph from the existing page-based call graph.

$$\textsc{FindPtrNode}(f_i, f_j) = \begin{cases} f_i & \text{if } size(f_i)+size(f_j) > \textsc{PageSize} \\ f_i \cup (f_j \in (\min(\{Eq.9 | f_j \in f_i\text{'s parent}\}, & \\ \quad \{Eq.10 | f_j \in f_i\text{'cousin}\}, & \\ \quad \{Eq.11 | f_j \in f_i\text{'s child}\})) & \text{if } size(f_i)+size(f_j) \le \textsc{PageSize} \end{cases} \quad (12)$$

5. PAGE-BASED IMAGING

The IMAGING procedure is invoked after CLUSTERING to create a page-based image and to generate a diff script to be disseminated over wireless networks. The flowchart for IMAGING is shown in the lower part of Fig. 3, and its pseudocode is shown in Fig. 9. The primary objective is to minimize the influence of code shift on references. Another objective is to minimize the size of the diff script that it generates. IMAGING is further decomposed into two procedures named ORDERING and GENDIFF.

5.1 Ordering

ORDERING performs *intra-page* arrangement of functions. The purpose is to place those functions that are likely to be modified near the end of the page. This way, they will less likely disturb other functions within the same page, because only functions placed after them can potentially be shifted.

As an illustration, Figs. 10(a) and (b) show two different images named Old Image 1 and Old Image 2 for the same initial version of the program. The difference is that in Page 2, the former arranges the function F_2 before F_3 while the latter does F_3 before F_2. The point of this example is to show that a good initial ordering even just within Page 2 can lead to dramatically lower perturbation to the code memory, when function F_2 is enlarged.

Starting with Old Image 1, Fig. 10(a) may evolve into either New Image 1 or New Image 2, depending on how the enlarged function F_2 is kept in the original page (Page 2) or put in a newly allocated page (Page 5), respectively. If in the same page, F_2 still has the same starting address and therefore none of its callers need to change, but F_3 is shifted and all of its callers must be updated, including F_4 on Page 3 and F_5 on Page 4. In total, three pages must be updated. On the other hand, if the enlarged function F_2 is placed in a newly allocated Page 5, callers of F_2 need to be updated, and they also affect three pages (1, 2, 5) as shown in New Image 2, but it uses a total of five pages instead of four as New Image 1.

```
RECLUSTERING($DSS, PBCG$)
URPBI ← NULL
while $DSS \neq \{\}$
    do $TN \leftarrow DSS.pop()$
        $NS.push(TN)$
        $NS, URPBI \leftarrow$ RECURSIVECLUSTRING($NS, TN,$
                                   $URPBI, PBCG$)
return $URPBI$
```

```
RECURSIVECLUSTERING($NS, AncN, N, PtrN, URPBI, PBCG$)
if $NS \neq$ NULL
  then if $AncN =$ NULL
         then $AncN \leftarrow N$
              $N \leftarrow NS.pop()$
              $PreSv \leftarrow$ NULL
         else $AncN \leftarrow N$
              $N \leftarrow PtrN$
              $PreSv \leftarrow CurSv$
       $CurSv, PreSv, PtrN$
           $\leftarrow$ FINDPTRNODE($AncN, N, PBCG, PreSv$)
       if $CurSv =$ NULL
         then $NS.push(N)$
              RECURSIVECLUSTRING($NS, AncN, N, PtrN, URPBI$
                    $PBCG$)
         else $URPBI, PBCG$
              $\leftarrow$ CLUSTERINGTWONODES($N,$
                    $PtrN, URPBI, PBCG$)
return $NS, URPBI$
```

Figure 8: RECLUSTERING **and** RECURSIVECLUSTERING **pseudocode.**

```
IMAGING($DSS, URPBI, OPBI$)
$PBI \leftarrow$ ORDERING($URPBI$)
$DS \leftarrow$ GENDIFF($DSS, OPBI, PBI$)
return $DS, PBI$
```

Figure 9: IMAGING **algorithm.**

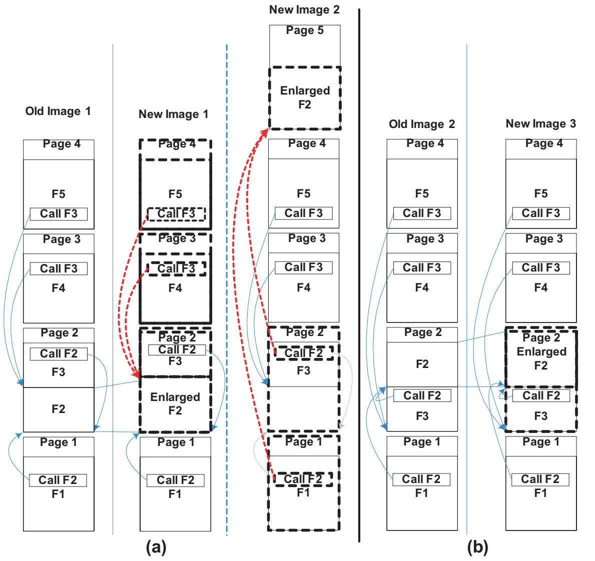


Figure 10: Different Layouts and Different Updates. (a) In case of the enlarged function, F_2 placed at lower address than F_3. (b) In case of F_2 placed at higher address than F_3.

Fig. 10(b) shows that a different initial image (Old Image 2) can reduce the number of affected pages from three down to one, simply by ordering F_3 before F_2 on Page 2. The function F_2 can be enlarged within Page 2 without affecting the starting address of either F_2 or F_3. Therefore, none of their callers need to be updated, and the only page that needs to be updated is Page 2.

How does one determine what functions are more likely to be modified than others? Several software metrics can be considered, including the number of lines of the source code, Cyclomatic complexity [9], and code coverage have been proposed. It has been reasoned that a function with higher logical complexity is more likely to contain errors and therefore more likely to require bug fixes [10], and it can be quantified by the Cyclomatic Complexity metric.

5.1.1 *Ordering Determination*

Calculating the influence of each function involves evaluating the likelihood of change. The influence can be derived by the complexity of each function and the number of references to each function. The complexity can be measured by using cyclomatic complexity[9] based on analyzing its control flow graph. The cyclomatic complexity counts the number of linearly independent paths of each function in order to obtain its quantitative values. The equation of the Cyclomatic Complexity is as follows:

$$M = E - N + 2P \tag{13}$$

where

$M =$ cyclomatic complexity
$E =$ the number of edges in the graph
$N =$ the number of nodes in the graph
$P =$ the number of connected components.

We use the tool called C & C++ Code Counter (CCCC) [11] to obtain the quantitative value of cyclomatic complexity.

To quantify the influence of a function on other functions, we define the Influence Equation $IE(f_k)$ of the function f_k as the Cyclomatic Complexity of f_k weighted by the number of references to f_k, as shown below:

$$IE(f_k) = CC(f_k) \times NR^{f_m}(SF - \{f_k\}) \tag{14}$$

where $CC(f_k)$ denotes the Cyclomatic Complexity of the function, SF is a set of functions $\{f_1, f_2, f_3, ..., f_n\}$ within a page, and $NR^{f_k}()$ is the number of references to f_k.

5.1.2 *Ordering Algorithm*

Equation (20) calculates the influence value of each function within each page. The influence value is based on the complexity of each function within each page and total number of references to the function. Consequently, we use the influence value to determine which function is more likely to be modified relative to others in the same page and should be placed towards the end of the page. The ORDERING procedure ranks each function based on the influence vales. The sets named *SF*, *SC*, and *SR* are defined as follows.

$$SF = \{f_1, f_2, f_3, ..., f_n\} \tag{15}$$

$$SC = \{c_1, c_2, c_3, ..., c_n\} \tag{16}$$

$$c_k = CC(f_k) \tag{17}$$

$$SR = \{r_1, r_2, r_3, ..., r_n\} \tag{18}$$

$$r_k = NR^{f_k}(SF - \{f_k\}) \tag{19}$$

$$IE(f_k) = \sum_{k=1}^{n} (SR - \{r_k\}) \times c_k \tag{20}$$

where *SF* is a set of functions within a page, *SC* is a set of the complexity of the functions, and *SR* is a set of the numbers of references to the functions. $IE(f_k)$ calculates the influence value of function f_k. Note that *SF* is for unsorted functions while the sequence *NSF* is for sorted functions. After calculating the influence values among unsorted functions one by one, a function having the minimum influence value is moved from *SF* to *NSF*. ORDERINGWITHINPAGE(*SF*) ranks each function within each page to resolve each function's start address.

```
ORDERING(URPBI)
for $k \leftarrow 0$ to NUMBEROFPAGE
    do $Page_k \leftarrow$ GETTINGPAGE($k$)
       $Page_k \leftarrow$ ORDERINGWITHINPAGE($Page_k$)
       $PBI \leftarrow$ PAGEBASEDIMAGE($Page_k, PBI$)
return $PBI$
```

Figure 11: ORDERING **algorithm.**

```
ORDERINGWITHINPAGE(SF)
$NSF \leftarrow \{\}$
$ie_{min} \leftarrow \infty$
while $SF \neq \{\}$
    do for each element $f_k \in SF$
          do $ie_k \leftarrow IE(f_k)$
             if $ie_{\min} > ie_k$
                then $ie_{\min} \leftarrow ie_k$, $f_{\min} \leftarrow f_k$
       $SF \leftarrow SF \setminus \{f_{\min}\}$
       $NSF \leftarrow NSF \cup \{f_{\min}\}$
return $NSF$
```

Figure 12: ORDERINGWITHINPAGE **algorithm.**

5.2 Diff Scripting

The IMAGING procedure generates a diff script to be disseminated over the wireless link to the nodes. Issues with dissemination include network protocol design and security, though they are outside the scope of this work. Our diff script is similar to the previous work such as [1] in that it includes three primitives: insert, replace, and copy. The insert and replace primitives have the format of a one-byte opcode and two-byte destination address with *n* bytes of data or instructions. The format of the copy primitive is one byte of opcode, two bytes of source address, two bytes of destination address, and two bytes of length of the data or instruction block copied.

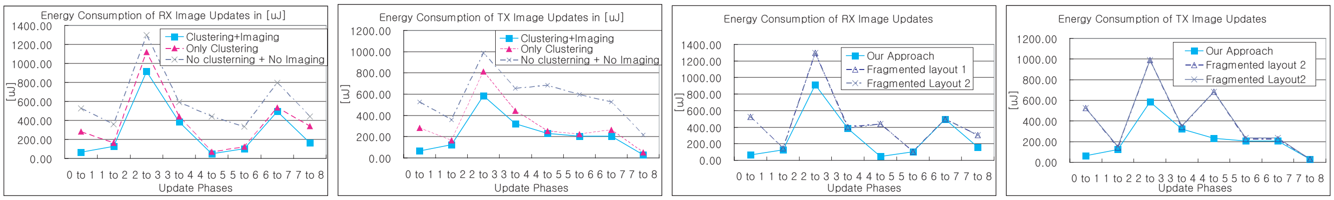
6. EXPERIMENTAL RESULTS

Table 3 shows our two test cases: (1) nine versions of RX (receive) and nine versions of TX (transmit). These images are compiled by the Small Device C Compiler (SDCC) [12] targeting Eco, an ultra-compact wireless sensor platform. We use the energy characteristics of NOR flash memory [8] with 128-byte pages.

We compare results from two groups of techniques on the two test cases. The first group consists of (a) CLUSTERING and IMAGING; (b) CLUSTERING with reversed ORDERING while in IMAGING; (c) no CLUSTERING and no ORDERING but only DIFF while in IMAGING only. The purpose of the first group of comparisons is to show the importance of page-based image layouts for flash memory. The second group consists of (a) our approach (CLUSTERING and IMAGING), (b) fragmented layouts with slop spaces, and (c) fragmented layouts by placing functions to free spaces to reduce

Table 3: Size of RX and TX mode images[byte]

Version	0	1	2	3	4	5	6	7	8
RX Image	3211	3194	3032	2445	2816	2838	2816	2537	2791
TX Image	3211	3194	3032	1912	2247	2282	2247	1924	1957



(a) Energy Consumption of RX Images

(b) Energy Consumption of TX Images

(c) Energy Consumption of RX Images by comparing with other flash-based image layouts

(d) Energy Consumption of TX Images by comparing with other flash-based image layouts

Figure 13: Energy consumption comparisons.

Table 4: Total Power Consumption of RX and TX Images through Progressive Firmware Update in [*uJ*]

	clustering+imaging	clustering+rev. order	no clustering+no imaging
RX	2290.02	3072.48	4784.64
TX	1769.23	2491.28	4552.08

Table 5: Total Power Consumption of RX and TX Image Updates by comparing with other flash-based image layouts in [*uJ*]

	clustering+imaging	fragmented layout 1	fragmented layout 2
RX	2290.02	3709.67	3689.84
TX	1769.23	3165.70	3205.35

code shift incidents. The purpose of the second group is to show the advantages of our layouts.

Among techniques in the first group, our approach results in lowest energy consumption by saving 25.28% and 52.02% energy for the RX and 38.27% and 37.94% for the TX. Among those in the second group, our technique saves 41.11% and 41.80% energy for RX and 31.4% and 37.9% for the TX over both other fragmented layouts approaches.

7. CONCLUSION

This paper proposes a compile-time, page-based code-layout technique for remote firmware update for NOR-flash-based embedded systems. The series of code images generated by our technique evolves well by minimizing not only the size of the *diff* scripts between successive versions, but also the amount of patching. Both result in low energy consumption. These properties are achieved by CLUSTERING, which performs grouping of functions in page-size partitions to minimize inter-page influence, and by IMAGING, which orders the functions within a page to minimize intra-page influence that turns into inter-page ones. We quantify the influence by not only caller-callee relationship but also Cyclomatic Complexity to predict the likelihood of change. Experimental results show our technique to consistently yield not only significantly lower energy consumption than state-of-the-art ones but also minimizes wear and tear of the NOR flash memory.

Acknowledgments This work was sponsored in part by National Science Foundation Grants CNS-0712926 and CNS-0448668. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Appendix: List of Symbols used in Algorithms

AncN:	ancestor (parent) node
CurSv:	current energy save
DS:	diff script
DSS:	data structure set of modified functions
f_k:	the k^{th} function in *DSS*
MF:	modified functions
NS:	stack for nodes
NSC:	new source code
NSF:	new sequence of functions
N:	a node
OPBI:	old page-based image
OS:	old source code
PBCG:	page-based call graph
PBI:	page-based image
$Page_k$:	the k^{th} page
PreSv:	previous energy save
PrtN:	partner node (parent, cousin, or child)
SF:	unsorted functions
TN:	temporary node
URPBI:	unresolved page-based image
CLUSTERINGTWONODES():	clusters the node with the parter node
ENFCOST():	energy cost for an enlarged function
FINDPTRNODE():	finds the parter node
RMFCOST():	energy cost for a removed function
SHFCOST():	energy cost for a shrunk function

8. REFERENCES

[1] N. Reijers and K. Langendoen. Efficient code distribution in wireless sensor networks. In *In Proceedings of the Second ACM International Workshop on Wireless Sensor Networks and Applications (WSNA '03)*, pages 60–67, 2003.

[2] Jaein Jeong and D. Culler. Incremental network programming for wireless sensors. In *Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004. 2004 First Annual IEEE Communications Society Conference on*, pages 25–33, Oct. 2004.

[3] Pedro José Marrón, Matthias Gauger, Andreas Lachenmann, Daniel Minder, Olga Saukh, and Kurt Rothermel. FlexCup: A flexible and efficient code update mechanism for sensor networks. In *EWSN 2006*, pages 212–227, February 2006.

[4] Weijia Li, Youtao Zhang, Jun Yang, and Jiang Zheng. UCC: update-conscious compilation for energy efficiency in wireless sensor networks. In *Proceedings of the 2007 PLDI conference*, volume 42, pages 383–393, 2007.

[5] J. Koshy and R. Pandey. Remote incremental linking for energy-efficient reprogramming of sensor networks : Wireless sensor networks. In *Proceeedings of the Second European Workshop*, pages 354–365, Jan.–Feb. 2005.

[6] Carl von Platen and Johan Eker. Feedback linking: optimizing object code layout for updates. In *Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Language, Compilers, and Tool Support for Embedded System (LCTES)*, pages 2–11, July 2006.

[7] Eco mote. `http://ecomote.net/`.

[8] AT29C010A full data sheet :1-megabit 5-volt only flash memory. `http://www.atmel.com/dyn/resources/prod_documents/doc0394.pdf`.

[9] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, December 1976.

[10] Todd L. Graves, Alan F. Karr, J. S. Marron, and Harvey Siy. Predicting fault incidence using software change history. In *IEEE Transactions on Software Engineering*, pages 100–108, 1999.

[11] CCCC – C and C++ Code Counter. `http://cccc.sourceforge.net/`.

[12] Small Device C Compiler. `http://sdcc.sourceforge.net/`.

[13] rsync. `http://rsync.samba.org/rsync/`.