Energy-Efficient Progressive Remote Update for Flash-Based Firmware of Networked Embedded Systems

Jinsik Kim
University of California, Irvine
Irvine CA, USA 92697-2625
jinsikk@uci.edu
and
Pai H. Chou
University of California, Irvine, CA,92697-2625, USA and
National Tsing Hua University, Hsinchu, 30013 Taiwan
phchou@uci.edu

Firmware update over a network connection is an essential but expensive feature for many embedded systems due to not only the relatively high power consumption and limited bandwidth, but also page-granular erasure before rewriting to flash memory. 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. We propose a tool that first clusters functions to minimize caller-callee dependency across pages, and then orders the functions within each page to minimize intra-page perturbation. Experimental results show our technique to reduce the energy consumption of firmware update by 30–42% over the state-of-the-art. Most importantly, this is the first work that has ever shown to evolve well over 41 revisions of a real-world open-source real-time operating system.

Categories and Subject Descriptors: D.3.4 [**Processors**]: Code generation

General Terms: Algorithms, Management, Measurement, Performance

Additional Key Words and Phrases: High-level analysis, NOR Flash memory, Page, Diff, Clycomatic complexity, Progressive code update, Embedded systems

The authors' contact address is: University of California, Irvine, CA 92697-2625, USA. Email {jinik, phchou}@uci.edu.

A previous version of this paper appeared as Jinsik Kim and Pai H. Chou, "Remote Progressive Firmware Update for Flash-Based Networked Embedded Systems," in *Proc. International Symposium on Low Power Electronics and Design* (ISLPED), San Francisco, CA, USA, Aug. 19-21, 2009, pp. 407–412. This manuscript contains a significant amount of new materials, including mathematical formulation for more rigorous definitions as well as all new experimental results on 42 revisions of FreeRTOS, an open-source embedded operating system.

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. Another increasingly important reason is runtime optimization with dynamic compilation by the host. For instance, if a variable is determined to be a runtime constant, then it can be compiled as more efficient code without many conditional statements or unnecessary instructions. By patching the code at runtime, the resulting system will be able to run faster using less energy. Of course, the premise is that the patching overhead is kept low.

Firmware is usually stored in nonvolatile memory such as EEPROM or flash. Remote firmware update can be an expensive process for many embedded systems, especially considering the potential application to dynamic compilation. In many systems that have been designed to date, RF transceivers and flash memory consume more power than other components by a wide margin. 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. For dynamic compilation, an additional requirement is that the patched code evolves well over the *entire lifetime* of the system. Existing techniques may leave gaps intentionally to avoid shifts, but their effectiveness over time has not been demonstrated.

We propose a new technique called Remote Progressive Firmware Update (RPFU), which improves over the state-of-the-art by considering the characteristics of different functions in not only grouping them in the same pages but also ordering them within each page. This is a step performed during linking after compilation. The resulting code image translates into a small patching script to minimize energy for transmission. Moreover, the patching script performs minimal shifting, thereby reducing the number of unnecessary rewrites to the flash memory. A distinguishing property of our technique is that it *evolves* well over the entire lifetime of the system, not just between some randomly chosen pair of successive revisions. Note that dynamic compilation is only one of many possible applications of this work but not the primary one. Our contribution is much more general, and we show the effectiveness of our technique over 8 to 41 consecutive revisions of real applications.

2. RELATED WORK

Previous works have studied 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 revisions [Reijers and Langendoen 2003; Jeong and Culler 2004; Marrón et al. 2006; Li et al. 2007]. They have the effect of reducing communication cost but do not consider additional cost of page writes when there are references across different pages. 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 [Koshy and Pandey 2005] 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 [von Platen and Eker 2006] takes a 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. It minimizes not only the difference between two arbitrary successive revisions but also the total Hamming distance from the first to the last revision. This means it will be not only energy efficient to transfer, since the patching script is small, but also energy efficient to patch, since the shifting and rewriting are minimized.

3. PROBLEM STATEMENT

The concept of a page is central to the operation of different types of flash memory. Unlike EEPROM or RAM, which are byte-writable, flash memory must be erased in units of pages before it can be rewritten. This property is true for both NAND and NOR types of flash memory. In this paper, the erasure unit of NOR flash memory is called a *page*, even though the term *segment*, *block*, or *sector* may be used in the data sheet. Mathematically, a flash memory system can be defined as an array of *physical pages* $\Phi = \langle \phi_1, \phi_2, \ldots, \phi_{np} \rangle$, where all pages are of the same size M. Because physical pages are to contain executable code after the linker has performed the linking step, they are also called *post-linking pages*, to be distinguished from *pre-linking* pages to be defined later.

Functions that define the firmware in the flash memory are representations of code units before the linker stage. A *(pre-linking) function* is denoted by f_i for $i \in A$, where $A = \{1, 2, \ldots, n\}$ is the *index set*. The *set of (pre-linking) functions* is denoted by $F = \{f_1, f_2, \ldots, f_n\}$. An *array of (ordered, pre-linking) functions* is denoted by $F' = [f'_1, f'_2, \ldots, f'_n]$, where $f_i = f'_j$ such that $j \in A \wedge o(i) = j$, where $o : A \to A$ defines a linker order for the array F'. While $O(f_i) \triangleq F'[o(i)] \leftarrow f_i$ is defined as a function that places a function in the set F into the j^{th} position in the array F', $O^{-1}(f_i) \triangleq f'_j \leftarrow F'[o(i)]$ is defined as a procedure that obtains an element in the array F' corresponding to a function f_i in the set F.

In contrast, *function images* can be defined as models that are outputs of the linker and consist of strings of bytes. A *(post-linking) function image* is denoted by β_i in terms of a byte string, and an *array of (post-linking) function images* is denoted by $B = [\beta_1, \beta_2, \ldots, \beta_n]$. The *firmware image* β after linking functions in the firmware is the concatenation of function images separated by *holes*, or padded space. That is, $\beta = \beta_1 \cdot d_1 \cdot \beta_2 \cdot d_2 \cdot \ldots \cdot \beta_n \cdot d_n$, where d_i is padded space between β_i and β_{i+1}.

The firmware is represented by a set of functions F, and some subsets of F are clustered into pages in a set, which is named a *pre-linking page set*: $\Pi = \{\pi_1, \pi_2, \ldots, \pi_{np}\}$, where $\pi_i \in 2^F, \bigcup_{1 \le i \le np} \pi_i = F$, and $\pi_i \cap \pi_j = \emptyset \, \forall i \neq j$ and $i, j \in [1, n]$.

The procedure $\textsc{Cluster}(F, \Pi) \triangleq \Pi'$ takes a set of subsets denoted by *SSF*, i.e., $SSF = \{F_j | F_j \in 2^F \wedge \bigcup_{1 \le j \le ppn} F_j = F \wedge |F_j| \le M\}$, and clusters all functions of each subset into a pre-linking page set denoted by Π', i.e., $\Pi' = \{\pi'_j | \pi'_j \in \Pi \wedge \bigcup_{1 \le j \le ppn} \wedge \pi'_j = F\}$, where $ppn \le n \le pn$.

Functions in pre-linking pages are relocated to their specific positions in the image by a linker defined by the function composition of $O^{-1} : F \rightarrow F'$ and $\Lambda : F' \rightarrow B$, i.e., $L(f_i) = (\Lambda \circ O^{-1})(f_i)$. $(\Lambda \circ O^{-1})(f_i) = \beta_j$ maps i in the arbitrary order to o_j in the linking order as well as a function f_i to its image β_j for $i, j \in A$.

On the other hand, since a firmware image in a flash memory system is partitioned into multiple post-linking pages, a binary string in a post-linking page is denoted by $\langle \beta_h \cdot d_h \rangle$ that is one of the substrings of the firmware image β, where h is denoted by $\langle h | a \le h \le b, a \in A, b \in A \rangle$. We assume that every function image fits in a page, or else a preprocessing step partitions the function image. So, $|\beta_i| \le M \, \forall i$.

Firmware may be associated with a *revision number* $\gamma \in \Gamma = \{1, 2, \ldots, l\}$. We can qualify the functions and the corresponding images with a revision number using the parenthesized superscript notation, e.g., $F^{(\gamma)} = \{f_1^{(\gamma)}, f_2^{(\gamma)}, \ldots, f_n^{(\gamma)}\}$. Note that a function may be added, deleted, modified, but its index does not change over different revisions. Instead, its code size is set to zero before adding or when deleted. A *revision* is a concept for a build, similar to the Subversion (SVN) system, rather than a file-specific *version* as in Concurrent Versions System (CVS). This means $f_i^{(\gamma)}$ and $f_i^{(\gamma+1)}$ may be identical or entirely different. We may omit the revision number γ when it is understood or whenever convenient.

Furthermore, the code difference between the functions in successive revisions is denoted by $\Delta(f_i^{(\gamma)}, f_i^{(\gamma+1)})$ that generates a *segment information set of a function pair*, i.e., $S_i^{(\gamma)}$. In the ensuing text, we use $\Delta(f_i^{(\gamma)})$ as a shorthand for $\Delta(f_i^{(\gamma)}, f_i^{(\gamma+1)})$.

The *segment information set of a function pair* is defined as $S_i^{(\gamma)} = \{s_{i_k}^{(\gamma)} | k \in [1, ns_i^{(\gamma)}]\}$, where $ns_i^{(\gamma)}$ is the number of difference strings between $\beta_j^{(\gamma)}$ and $\beta_j^{(\gamma+1)}$, $s_{i_k}^{(\gamma)}$ consists of $(\mu_{i_k}^{(\gamma)}, \nu_{i_k}^{(\gamma)})$, $\mu_{i_k}^{(\gamma)}$ is the start address of the k^{th} difference string, $\nu_{i_k}^{(\gamma)}$ is the k^{th} string that is one of the slices of $\beta_j^{(\gamma+1)}$, and $|S_i^{(\gamma)}| \le |\beta_j^{(\gamma+1)}|$.

Based on $S_i^{(\gamma)}$, *a set of the properties of different image* is denoted by $DC^{(\gamma)}$, i.e., $DC^{(\gamma)} = \{dc_1^{(\gamma)}, dc_2^{(\gamma)}, \ldots, dc_{ns}^{(\gamma)}\}$, where $dc_i^{(\gamma)}$ corresponding to $\mu_i^{(\gamma)}$ consists of an attribute ATT, source address SRC, and destination address DST, i.e., $dc_i^{(\gamma)} = (ATT_i^{(\gamma)}, SRC_i^{(\gamma)}, DST_i^{(\gamma)})$. An attribute can be either "copy", "replace", or "insert".

$\Delta(f_i^{(\gamma)})$ is defined by a function composition of $\delta(\beta_i^{(\gamma)}, \beta_i^{(\gamma+1)}) = S_i^{(\gamma)}$ and $O^{-1}(f_i) = \beta_j$, i.e., $\Delta(f_i^{(\gamma)}) = \delta(O^{-1}(f_i^{(\gamma)}), O^{-1}(f_i^{(\gamma+1)}))$, where $\delta(\beta_i^{(\gamma)}) = S_i^{(\gamma)}$ maps two firmware images to the set $S_i^{(\gamma)}$, and $\delta(\beta_i^{(\gamma)})$ is a convenient expression of $\delta(\beta_i^{(\gamma)}, \beta_i^{(\gamma+1)})$. In this paper, we define $||\Delta(f_i^{(\gamma)})|| = \sum_{k=1}^{|S_i^{(\gamma)}|} |\mu_{i_k}^{(\gamma)}|$ to be the *total number of bytes of code difference.*

The set of function pairs is defined as $FF^{(\gamma)} = \{(f_i^{(\gamma)}, f_i^{(\gamma+1)}) | 1 \leq i \leq n\}$. In addition, based on $\Delta(f_i^{(\gamma)})$, a set of modified functions between successive revisions can be extracted and denoted by $MF^{(\gamma)}$. Assuming $f_i^{(\gamma+1)} = f_j'^{(\gamma+1)}$:

—if $|\beta_j^{(\gamma)}| < |\beta_j^{(\gamma+1)}|$, then $f_i^{(\gamma+1)}$ is an *enlarged* function;

—if $|\beta_j^{(\gamma)}| \geq |\beta_j^{(\gamma+1)|}$, then $f_i^{(\gamma+1)}$ is a *shrunk* function;

—if $|\beta_j^{(\gamma+1)}| = 0$, then $f_i^{(\gamma+1)}$ is a *removed* function; and

—if $|\beta_j^{(\gamma)}| = 0$, then $f_i^{(\gamma+1)}$ is a *newly added* function.

Functions can be related to each other as callers and callees, as defined by the caller-to-callee pairs: $C \subseteq F \times F$. Several functions related to caller-to-callee pairs are defined as follows. $r : F \rightarrow 2^F$ maps a function to its callees, i.e., $r(f_i) = \{f_j | (f_i, f_j) \in C\}$, and $r^{-1} : F \rightarrow 2^F$ maps a function to its callers, i.e., $r^{-1}(f_i) = \{f_j | (f_j, f_i) \in C\}$. $R : 2^F \rightarrow 2^F$ maps the union of functions to their callees, i.e., $R(F_1) = \{r(f_i) | f_i \in F_1\}$ for $F_1 \in 2^F$ and $R(F_1) \subseteq 2^F$, and $R^{-1} : 2^F \rightarrow 2^F$ maps the union of functions to their callers, i.e., $R^{-1}(F_1) = \{r^{-1}(f_i) | f_i \in F_1\}$ for $F_1 \in 2^F$ and $R^{-1}(F_1) \subseteq 2^F$. While the function $p(f_i) = \pi_j : f_i \in \pi_j$ (uniquely) maps a function to the (pre-linking) page where it is located, $P(F_1) = \{p(f_i) | f_i \in F_1\}$ maps a set of functions $F_1 \in 2^F$ to the subset of the power set of pre-linking pages where those functions are located. In the same manner, these function P and p are applied to function images and post-linking images, i.e., $P : 2^B \rightarrow 2^\Phi$ and $p : B \rightarrow \Phi$. Furthermore, $P(r^{-1}(f_i))$ then denotes the subset of the set of pages that contain callers to f_i where f_i is the i^{th} function in a set of modified functions.

Modifying a function may induce additional modifications to other functions in two different ways: *in-place modification* and *page reassignment.* In the first case, modifying a function in-place may mean shifting code of other functions located after the modified function that is within the same page. This will in turn cause other pages containing callers to those shifted functions that are to be updated as well, and this represents the worst case of modification. In the second case, all pages containing callers to the function that is modified need to be updated.

If a callee must be shifted, then it is necessary to update all callers to the new address of the callee. After clustering functions in a page, they need to be properly ranked to reduce the code shift, since a different ordering results in a different amount of code shift due to the complexity, number of inter-page and intra-page references of each function. The number of inter-page and intra-page references is denoted by $NR(\{\{\beta_i\}\}, \Pi \backslash \{\phi_i\}) + NR(P(R(\{\beta_i\}), P(R(\phi_i)))$, where $NR : 2^\beta \times$

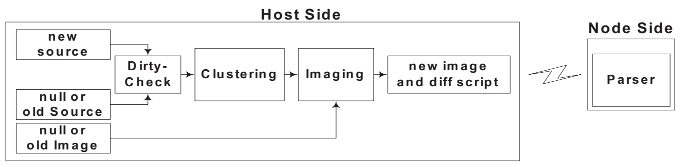


Fig. 1. Framework Block Diagram

$2^{\beta} \rightarrow \mathbb{Z}$ and $\beta_i \in \phi_i$. We denote that $NR(PB_1, PB_2) = |\{\beta_x | B_1 \in PB_2 \wedge f_x \in B_1 \wedge B_2 \in PB_2 \wedge \beta_x \in B_2\}|$, where $PB_1 \subseteq 2^{\beta}$ contains callers, and $PB_2 \subseteq 2^{\beta}$ contains callees.

The number of pages to be modified is equal to $|P(\beta_j^{(\gamma+1)})| + |P(r^{-1}e(\beta_j^{(\gamma+1)}))|$ such that $f_i^{(\gamma+1)} \in MF^{(\gamma)}$. The number of pages to be modified varies according to the different order of functions within the page, because the number of inter-page references to be modified depends on the number of callees to be shifted, i.e., $|R^{-1}(e(\beta_j^{(\gamma+1)}))|$, and the amount of code to be shifted, i.e., $|P(e(\beta_j^{(\gamma+1)}))|$, is different from the order of the functions that are within the page, because there are $|\pi_i|!$ different orderings in a page π_i, where $M\Pi^{(\gamma)} = \{m\pi_k^{(\gamma)} | k \in [1, mp]\}$, and mp is the number of pages to be modified, and $e(\beta_j^{(\gamma+1)})$ outputs a set of shifted functions that are located at higher positions than $\beta_j^{(\gamma+1)}$.

In summary, since flash memory has the different characteristic from byte-writable memories, firmware should be partitioned into pages with maximizing multiple modifications into pages, minimizing additional updates caused by code shift.

4. TECHNICAL APPROACH

Our proposed approach is called RPFU, for remote progressive firmware update. Fig. 1 shows the the top-level algorithm, which calls two procedures named Clustering and Imaging for the purpose of generating code images and patching scripts.

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 patching script. The patching 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 patching script is parsed and executed on the deployed node side.

4.1 Assumptions

We make several assumptions. First, this method applies to updates of monolithic binaries for real-time system platforms without memory management units. The updates are through a communication interface, which is relatively costly to operate in power consumption (e.g., RF module of a wireless sensor node) and relatively slow. In addition, the requirement of nonvolatile memory to erase a page further adds to the energy and time costs of code updating processes.

Flash memory is widely used in networked embedded systems including mobile ones [Park et al. 2004]. While flash memory can be of a NOR or NAND type, we

```
RPFU($NSC$, $OSC$, $OPBI$)
1  $MF \leftarrow$ DirtyCheck($NSC$, $OSC$)
2  $MF2 \leftarrow copy(MF)$
3  $URPBI \leftarrow$ Clustering($MF$, $OPBI$)
4  $DFFS, PBI \leftarrow$ Imaging($MF2$, $URPBI$)
5  return $DFFS, PBI$
                    (a)
```

```
Clustering($MF$, $OPBI$)
1  $DSS, PBCG \leftarrow$ Splitting($MF$, $OPBI$)
2  $URPBI \leftarrow$ Reclustering($DSS$, $PBCG$)
3  return $URPBI$
                    (b)
```

```
Imaging($DSS$, $URPBI$, $OPBI$)
1  $PBI \leftarrow$ Ordering($URPBI$)
2  $DFFS \leftarrow$ GenDiff($DSS$, $OPBI$, $PBI$)
3  return $DFFS, PBI$
                    (c)
```

Fig. 2. (a) Top-level algorithm. (b) Clustering algorithm. (c) Imaging algorithm.

assume the use of a NOR-type flash memory for firmware execution in place due to its ability to perform byte-reading and page-erasing. Also, it is a more popular and simpler form of nonvolatile program memory for embedded systems.

4.2 Algorithm

Clustering and Imaging for the purpose of generating code images are shown in Fig. 2(a), RPFU() pseudocode. The symbols in all pseudocode in this paper are listed in the Appendix. The Clustering algorithm is shown as pseudocode and as a flow chart in Figs. 2(b) and 3. The Splitting and Reclustering procedures are presented next. The pseudocode for Imaging is shown in Fig. 2(c). The primary objective is to minimize the influence of code shift on references, and then to minimize the size of the patching script that it generates. Imaging is further decomposed into two procedures named Ordering and GenDiff.

5. 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, or $NR(\Pi, \Pi)$), is minimized. In other words, the Clustering procedure minimizes the number of references crossing pages, and then it generates an unresolved (pre-linking) page-based image Π' .

The Clustering procedure is further divided into Splitting and Reclustering. Splitting extracts a call graph structure for the functions in the input source code. In this paper, the call graph is named a *page-based-clustering call graph* denoted by $G(V, E, H)$, where V is the set of vertices, $E \subseteq V \times V$, and $H \subseteq \Pi$. Some of the vertices are grouped by pages while the rest are split from pages in that they represent the modified functions.

If this is the very first version of the program, then the graph covers the entire set of function images. Otherwise, it covers the set of modified function images that are the user-modified ones as well as callee-induced modified ones ($\beta_i^{(\gamma-1)} \neq \beta_i^{(\gamma)}$), plus those in an existing page-based image. The call graph structure is fed to the next step, Reclustering. The purpose of Reclustering is to create either a

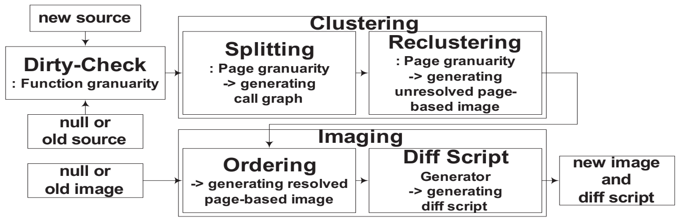


Fig. 3. Framework on the host side in detail

good initial grouping or minimally different grouping that will result in low energy consumption when transmitted and updated. Each group of functions will fit within a page and then ordered to further minimize intra-page shifts.

5.1 Splitting

When Splitting is being performed, it is necessary to estimate the influence of relocating a modified function on those pages that contain references to the modified function. The number of page-crossing references is defined by Equation (1). Since Splitting can split the modified function in diverse ways, it should determine which way causes a code update to consume less energy.

Different ways to split the modified function can lead to different numbers of referring pages to be modified, which is equal to the cardinality of the set (union) of pages that contain callers to the given function, i.e., $P(r^{-1}(f_i))$ that is defined as follows.

$$P(RFS) = \{\pi_i | f_j \in RFS \cap \pi_i \wedge (f_j, f_i) \in C\}, \text{ where } RFS = r^{-1}(f_i). \tag{1}$$

As compared to an exiting prior image, a function is considered modified if it is enlarged, shrunk, removed, or newly added because those functions have their own splitting options. For example, in Fig. 4, $\beta_4^{(\gamma)}$ is enlarged and renamed as $f_4^{(\gamma+1)}$ before linking.

One option is to write (the image of) $f_4^{(\gamma+1)}$ to a newly assigned (physical page of) $\pi_5^{(\gamma+1)}$, which necessitates updates to $\beta_4^{(\gamma)}$'s callers on $\phi_2^{(\gamma)}$. $\pi_5^{(\gamma+1)}$ and $\phi_2^{(\gamma+1)}$ then become the pages into which the modified functions would be clustered during Reclustering.

Another option is to write (the image of) $f_4^{(\gamma+1)}$ back to $\phi_3^{(\gamma+1)}$ by shifting $\beta_3^{(\gamma)}$. Although this does not affect (the image of) the callers of $\beta_4^{(\gamma)}$, it affects (the image of) the callers of $\beta_3^{(\gamma)}$ and thus requires update to $\phi_1^{(\gamma)}$. $\phi_1^{(\gamma+1)}$ and $\phi_3^{(\gamma+1)}$ then become pages where modified functions would be clustered during Reclustering. Therefore, the "enlarged function," $f_4^{(\gamma+1)}$ (i.e., function whose image increased in size from one revision to the next), affects not only $\phi_1^{(\gamma+1)}$ and $\phi_2^{(\gamma+1)}$ but also $\phi_3^{(\gamma+1)}$, where $\beta_4^{(\gamma+1)}$ belongs. Among these influenced pages that would become $\pi_1^{(\gamma+1)}$, $\pi_2^{(\gamma+1)}$, and $\pi_3^{(\gamma+1)}$, respectively, Splitting should determine a way to minimize energy consumption for some of the pages that will be split and then clustered.

Consequently, 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 as Fig. 4

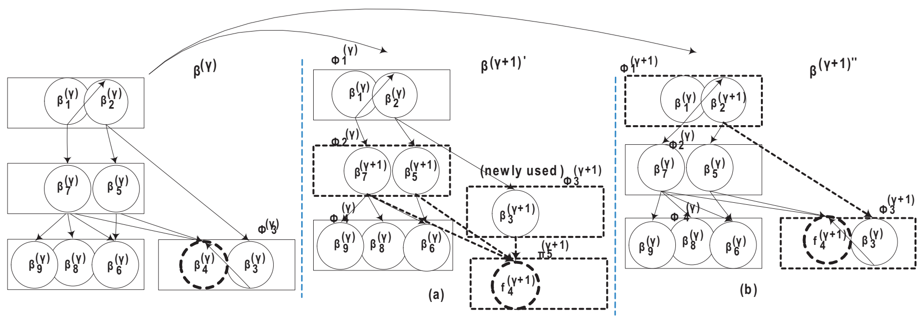


Fig. 4. Splitting in case of an enlarged function, $f_4^{(\gamma+1)}$. (a) the enlarged function, $f_4^{(\gamma+1)}$, moved to the ϕ_5 : ϕ_1, ϕ_3, and ϕ_5 to be updated. (b) $f_4^{(\gamma+1)}$ in place : ϕ_1 and ϕ_3 to be updated.

illustrates. A page needs to be updated if it contains either a modified function or references a relocated function. Note that a modified function may be of the same size, enlarged, shrunk, removed, or newly added with respect to the previous revision. The update style for each function can be further classified into

(1) *in-place* update, i.e., same starting address on the same page. Formally, $a(\beta_i^{(\gamma)}) = a(\beta_i^{(\gamma+1)}) \wedge |\beta_i^{(\gamma)} \cdot d_i^{(\gamma)}| = |\beta_i^{(\gamma+1)} \cdot d_i^{(\gamma+1)}|$, where $a(\beta_i)$ stands for the start address of β_i.

(2) *anew-in-place* update, which means updating one or more of the function images, i.e., $\beta_j^{(\gamma+1)}$, on the same page where *in-place* with $\beta_i^{(\gamma+1)}$ occurs at the same time. Formally, (*in-place* update condition) $\wedge a(\beta_i^{(\gamma+1)}) < a(\beta_j^{(\gamma+1)}) \wedge \beta_j^{(\gamma)} \neq \beta_j^{(\gamma+1)} \wedge p(\beta_i^{(\gamma+1)}) = p(\beta_j^{(\gamma+1)})$.

(3) writing the modified function $\beta_i^{(\gamma+1)}$ to free space, or *hole*, i.e., d_j, and $i \neq j$, in another page, i.e., $p(\beta_i^{(\gamma)}) \neq p(d_j^{(\gamma)})$. Formally, $a(\beta_i^{(\gamma+1)}) \geq a(d_j^{(\gamma)}) \wedge |\beta_i^{(\gamma+1)}| \leq |d_j^{(\gamma)}| \wedge p(\beta_i^{(\gamma)}) \neq p(d_j^{(\gamma)})$.

(4) *shifting* some other functions' images that are equal to $e(\beta_i^{(\gamma)})$ and $e(\beta_i^{(\gamma+1)})$ on the same page in addition to writing the modified function image $\beta_i^{(\gamma+1)}$. Formally, $a(\beta_i^{(\gamma)}) = a(\beta_i^{(\gamma+1)}) \wedge |\beta_i^{(\gamma)}| < |\beta_i^{(\gamma+1)}| \wedge a\left((\beta_k^{(\gamma)})_{\forall \beta_k^{(\gamma)} \in e(\beta_i^{(\gamma)})}\right) < a\left((\beta_k^{(\gamma+1)})_{\forall \beta_k^{(\gamma+1)} \in e(\beta_i^{(\gamma+1)})}\right)$.

(5) *anew-shifting* update, i.e., updating one of the shifted function images on the same page with *shifting* at the same time. Formally, (*shifting* update condition) $\wedge a(\beta_i^{(\gamma+1)}) < a(\beta_j^{(\gamma+1)}) \wedge \beta_j^{(\gamma)} \neq \beta_j^{(\gamma+1)} \wedge p(\beta_i^{(\gamma+1)}) = p(\beta_j^{(\gamma+1)})$,

(6) allocation of a *new* page for newly added functions, i.e., $|\beta_j^{(\gamma)}| = 0 \wedge |\beta_j^{(\gamma+1)}| \neq 0 \wedge \beta_j^{(\gamma+1)} \in \pi_j^{(\gamma+1)}$ or for modified function images that leave holes that have the same bytes as those of previous revision of the modified function images. Formally, $p(\beta_i^{(\gamma+1)}) \in \Phi^{(\gamma+1)} \wedge p(\beta_i^{(\gamma+1)}) \notin \Phi^{(\gamma)} \wedge p(\beta_i^{(\gamma)}) = \emptyset \wedge \beta_i^{(\gamma+1)} \neq \beta_i^{(\gamma)} \wedge a(\beta_i^{(\gamma+1)}) \neq a(\beta_i^{(\gamma)}) \wedge |\beta_i^{(\gamma)} \cdot d_i^{(\gamma)}| = |d_i^{(\gamma+1)}|$,

(7) *removing* a function image $\beta_i^{(\gamma)}$, i.e., $|\beta_j^{(\gamma)}| \neq 0 \wedge |\beta_j^{(\gamma+1)}| = 0$

The energy consumption for these update styles is modeled as Equations (2) through (8). On the right-hand side of the equations, the energy subscript refers to the component such as the flash memory, RAM, CPU, or an RF transceiver; and the parenthesized superscript, if present, refers to the task to perform on that component. For instance, $E_{\mathrm{CPU}}^{(\mathrm{RF})}(x)$ refers to the energy consumed by the CPU to perform the RF access of x units of data.

$$\begin{aligned} E_{\mathrm{inplace}}(f_i) = {} & (E_{\mathrm{CPU}}^{(\mathrm{flash+buf})}(|p(f_i)| \times M) + E_{\mathrm{buf}}^{(\mathrm{read+write})}(|p(f_i)| \times M) \\ & + E_{\mathrm{flash}}^{(\mathrm{read+erase+program})}(|p(f_i)| \times M) \\ & + E_{\mathrm{RF}}(||\Delta(f_i)||) + E_{\mathrm{CPU}}^{(\mathrm{RF})}(||\Delta(f_i)||) \end{aligned} \tag{2}$$

$$\begin{aligned} E_{\mathrm{anewinplace}}(f_i) = {} & (E_{\mathrm{CPU}}^{(\mathrm{flash+buf})}(|p(f_i)| \times M) + E_{\mathrm{buf}}^{(\mathrm{read+write})}(|p(f_i)| \times M) \\ & + E_{\mathrm{flash}}^{(\mathrm{read})}(|p(f_i)| \times M) \\ & + E_{\mathrm{RF}}(|R^{-1}(e(f_i))| + ||\Delta(f_i)||) + E_{\mathrm{CPU}}^{(\mathrm{RF})}(|R^{-1}(e(f_i))| + ||\Delta(f_i)||) \end{aligned} \tag{3}$$

$$\begin{aligned} E_{\mathrm{hole}}(f_i) = {} & E_{\mathrm{CPU}}^{(\mathrm{flash+buf})}(|p(f_i)| \times M) + E_{\mathrm{buf}}^{(\mathrm{read+write})}(|p(f_i)| \times M) \\ & + E_{\mathrm{flash}}^{(\mathrm{read+erase+program})}((|p(f_i)| + |P(R(f_i))|) \times M) \\ & + E_{\mathrm{RF}}(|r^{-1}(f_i)| + ||\Delta(f_i)||) + E_{\mathrm{CPU}}^{(\mathrm{RF})}(|r^{-1}(f_i)| + ||\Delta(f_i)||) \end{aligned} \tag{4}$$

$$\begin{aligned} E_{\mathrm{shift}}(f_i) = {} & E_{\mathrm{CPU}}^{(\mathrm{flash+buf})}(|P(e(f_i))| \times M) + E_{\mathrm{buf}}^{(\mathrm{read+write})}(|P(e(f_i))| \times M) \\ & + E_{\mathrm{flash}}^{(\mathrm{read+erase+program})}((|P(e(f_i))| + |P(r(f_i))|) \times M) \\ & + E_{\mathrm{RF}}(|R^{-1}(e(f_i))| + ||\Delta(f_i)||) + E_{\mathrm{CPU}}^{(\mathrm{RF})}(|R^{-1}(e(f_i))| + ||\Delta(f_i)||) \end{aligned} \tag{5}$$

$$E_{\mathrm{anewshift}}(f_i) = E_{\mathrm{RF}}(|R^{-1}(e(f_i))| + ||\Delta(f_i)||) + E_{\mathrm{CPU}}^{(\mathrm{RF})}(|R^{-1}(e(f_i))| + ||\Delta(f_i)||) \tag{6}$$

$$\begin{aligned} E_{\mathrm{new}}(f_i) = {} & E_{\mathrm{RF}}(|R^{-1}(e(f_i))| + ||\Delta(f_i)||) + E_{\mathrm{CPU}}^{(\mathrm{RF})}(|R^{-1}(e(f_i))| + ||\Delta(f_i)||) \\ & + E_{\mathrm{flash}}^{(\mathrm{read+erase+program})}(|P(r^{-1}(f_i))| \times M) \end{aligned} \tag{7}$$

$$\begin{aligned} E_{\mathrm{remove}}(f_i) = {} & E_{\mathrm{CPU}}^{(\mathrm{flash+buf})}(|P(r^{-1}(f_i))| \times M) + E_{\mathrm{buf}}^{(\mathrm{read+write})}(|P(r^{-1}(f_i))| \times M) \\ & + E_{\mathrm{RF}}(|r^{-1}(f_i)|) + E_{\mathrm{CPU}}^{(\mathrm{RF})}(|r^{-1}(f_i)|) \\ & + E_{\mathrm{flash}}^{(\mathrm{read+erase+program})}(|P(r^{-1}(f_i))| \times M) \end{aligned} \tag{8}$$

The symbols in Equation (2) to (8) in this paper are listed in the Appendix.

The SPLITTING algorithm inputs a set of modified functions, $MF^{(\gamma)}$, and the page-based (post-linking) image from the previous revision $\Phi^{(\gamma)}$. It first calls CG-DATASTRUCTURE to analyze the caller-callee relationship and the complexity of the functions, and then partitions them among the pages. In Fig. 5(b), VERTEX-DATASTRUCTURE establishes each function's data structure that consists of the function's name, size, complexity, the set of callees, and the set of callers in order to retain the caller-callee relationship and complexity of the functions. We assume that a preprocessing step divides larger functions into smaller ones that can each fit within a page. Then, SPLITTING calls PBCALLGRAPH to construct a page-based call graph (*PBCG*). The objective of SPLITTING is to minimize the cost of the *PBCG*.

```
SPLITTING($MF$, $OPBI$)
1  $DSS \leftarrow$ CGDATASTRUCTURE($MF$)
2  $DSS2 \leftarrow copy(DSS)$
3  $PBCG \leftarrow$ PBCALLGRAPH($DSS$, $OPBI$)
4  return $DSS2, PBCG$
```

(a)

```
CGDATASTRUCTURE($MF$)
1  $DSS \leftarrow [\,]$
2  while $MF \neq [\,]$
3      do $f_k \leftarrow MF.pop()$
4          $x \leftarrow$ VERTEXDATASTRUCTURE($f_k$)
5          $DSS.push(x)$
6  return $DSS$
```

(b)

```
PBCALLGRAPH($DSS$, $OPBI$)
1  $PBCG \leftarrow$ empty graph
2  while $DSS \neq [\,]$
3      do $x \leftarrow DSS$.dequeueVertexWithHighestFunctionAddress()
4          if $x.f_k$ is ENLARGEDFUNCTION
5              then UPDATECOSTFORENF($x$, $OPBI$, $\underline{PBCG}$)
6          elseif $x.f_k$ is SHRUNKFUNCTION
7              then UPDATECOSTFORSHF($x$, $OPBI$, $\underline{PBCG}$)
8              else UPDATECOSTFORRMF($x$, $OPBI$, $\underline{PBCG}$)
9  return $PBCG$
```

(c)

Fig. 5. (a) SPLITTING pseudocode. (b) CGDATASTRUCTURE pseudocode. (c) PBCALLGRAPH pseudocode. Underlined parameters are modified as a side effect.

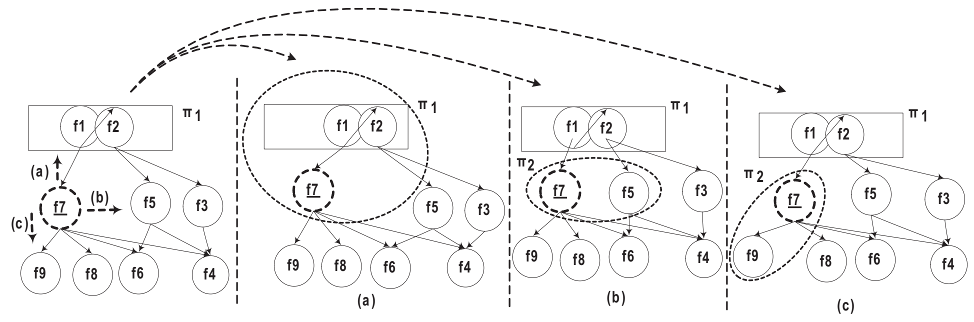


Fig. 6. Different ways of Clustering: (a) with a caller, (b) with an independent, (c) with a callee.

5.2 Reclustering

In terms of a caller-callee update, moving the start address of callees as a result of code shift causes the linker to modify the caller's image. To recall, to write even just a single byte to NOR flash requires erasing and rewriting the entire page. Therefore, the purpose of RECLUSTERING is to reduce the total number of page-granular accesses by folding multiple writes into one page rewrite through grouping callers and callees on the same page. In other words, maximizing the total number of intra-page references tends to minimize the number of inter-page references and therefore the total number of callee-induced page updates. This is simply because the total number of references consists of inter-page and intra-page ones:

$$|C| = \underbrace{NR(\Pi, \Pi)}_{\text{interpage}} + \underbrace{\sum_{i=1}^{np} NR(\pi_i, \pi_i)}_{\text{intrapage}} \tag{9}$$

where $|C|$ is the total number of references, $NR(\Pi, \Pi)$ is the total number of inter-page references, and $\sum_{i=1}^{np} NR(\pi_i, \pi_i)$ is the total number of intra-page reference. Of course, the number of callee-induced pages to modify need not be proportional to the total number of inter-page references, but in practice it is a good heuristic.

Fig. 6 shows the different ways that a set of functions from a call graph can be grouped. The call graph is generated after SPLITTING, where the edges represent caller-callee relationships and the vertices represent functions. Whenever RECLUSTERING is performed with one of the modified functions, it traverses the page-based-clustering call graph according to depth-first search (DFS) to find the page in which to cluster the modified function. RECLUSTERING repeats until the rest of the modified functions are clustered.

At the very first version, all of the functions are clustered into an image of prelinking pages $URPBI = \Pi'$, which is called an *unresolved page-based image.* The objective of RECLUSTERING is to ensure that $URPBI$ has the minimum number of references crossing pages that could potentially minimize the number of page-granular accesses. Then, starting from the first revision (second version), RECLUSTERING repeats the same way as the very first version except with modified functions.

RECLUSTERING adds the modified functions (MF) to the page-based call graph ($PBCG$) to generate an unresolved page-based image, $URPBI$. $UPBCG$ is defined as a graph $G(V, E, Q)$, where the vertices V represent F, the edges E represent C, Q represents a set of pages Π, and the vertices are grouped by the pages, and $PBCG$ is also defined as a graph $G(V, E, Q)$, where the vertices V represent β, the edges E represent C, Q represents pages Φ, and all vertices are clustered into the pages.

The objective of RECLUSTERING is to minimize the number of inter-page references. To do this, RECLUSTERING explores grouping functions that are related as (a) *callers* with a vertex, (b) *independents*, i.e., vertices with a common caller, and (c) the vertex with a subset of its *callees.* Fig. 6 shows an example of 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,

$$\text{minimize } \; NR(\Pi, \Pi) \tag{10}$$

Equations (11), (12), and (13) express the number of reference crossing pages ($NRCP$) after clustering with one of a caller, independent, and callee, respectively. RECURSIVECLUSTERING finds and merges a function into its caller, independent, or callee page.

```
RECLUSTERING($DSS$, $PBCG$)
1  $URPBI \leftarrow [\,]$
2  $VS \leftarrow [\,]$
3  while $DSS \neq [\,]$
4      do $x \leftarrow DSS.pop()$
5          $VS.push(x)$
6          RECURSIVECLUSTERING($\underline{VS}$, $\underline{URPBI}$, $PBCG$)
7  return $URPBI$
```

Fig. 7. RECLUSTERING pseudocode.

$$NRCP_{\text{caller}}(f_i, f_j) = NR(\Pi,\Pi) \setminus NR(\{f_j\},\{f_i\}),\ \text{where } f_j \in r(f_i) \tag{11}$$

$$NRCP_{\text{independent}}(f_i, f_j) = NR(\Pi,\Pi) \setminus NR(\{f_h\},\{f_i\}) \tag{12}$$
$$\text{where } f_i \in r^{-1}(f_h) \wedge f_j \in r^{-1}(f_h) \wedge f_h \in r(f_i) \cap r(f_j)$$

$$NRCP_{\text{callee}}(f_i, f_j) = NR(\Pi,\Pi) \setminus NR(\{f_i\},\{f_j\}),\ \text{where } f_j \in r^{-1}(f_i) \tag{13}$$

Based on Equation (14), FINDPARTNERVERTEX finds the vertex that will be clustered with the vertex x in Fig. 8. The found vertex is called a *PartnerVertex*, which is used as an input for CLUSTERINGTWOVERTICES. CLUSTERINGTWOVERTICES clusters x with *PartnerVertex* and adds the clustered vertex to an unresolved page-based call graph *UPBCG*, from the existing page-based call graph.

$$\text{FINDPARTNERVERTEX}(f_i, f_j) =$$
$$\begin{cases} f_i & \text{if } |f_i| + |f_j| > \text{M} \\ f_j \in r(f_i) & \text{if } Eq.11 = MM \wedge |f_i| + |f_j| \leq \text{M} \\ f_j \in r^{-1}(f_h) \wedge f_h \in r(f_i) \cap r(f_j) & \text{if } Eq.12 = MM \wedge |f_i| + |f_j| \leq \text{M} \\ f_j \in r^{-1}(f_i) & \text{if } Eq.13 = MM \wedge |f_i| + |f_j| \leq \text{M} \end{cases} \tag{14}$$

, where $MM = \min(Eq.\ (11), Eq.\ (12), Eq.\ (13))$, and M is the size of a page.

Consequently, the RECURSIVERECLUSTERING finds the clustering that can minimize the number of references, $NR(\Pi,\Pi)$, through traversing the page-based call graph according to *DFS*. To illustrate this algorithm, let us consider the example in Fig. 6. There, when considering the *CurrentVertex* f_7, the *PreVertex* is its caller f_1. When *CurrentVertex* is used as an input for ESTIMATEPTRVERTEX. It determines which partner vertex (which can be f_1, f_5, or f_9 in the example) is proper to be combined with *CurrentVerex* in order to consume less energy. The determination is based on Equation (14).

While *CurrentSave* is the number of references to be saved at the current reclustering step, *PreSave* is the number of references to be saved at the prior reclustering

```
RecursiveClustering(VS, URPBI, PBCG)
 1  if VS ≠ [ ]
 2     then x ← VS.pop()
 3           if x.PreVertex is empty
 4              then x.PreVertex ← x.CurrentVertex ▷ traverse to callee
 5                   x.CurrentVertex ← GetCallee(x) ▷ from a set, one callee at a time
 6                   x.PreSave ← Null
 7              else x.PreVertex ← x.CurrentVertex
 8                   x.CurrentVertex ← x.PartnerVertex
 9                   x.PreSave ← x.CurrentSave
10           x.CurrentSave, x.PreSave, x.PartnerVertex
                 ← EstimatePtrVertex(x, PBCG, PreSv)
11           if x.CurrentSave is empty
12              then VS.push(x)
13                   RecursiveClustering(VS, URPBI, PBCG)
14              else ClusteringTwoVertices(VS, URPBI, PBCG)
15  ▷ return VS, URPBI as side effect
```

Fig. 8. RecursiveClustering pseudocode.

step. x is the vertex currently being evaluated, while *PartnerVertex* is a vertex that would be clustered with x.

6. PAGE-BASED IMAGING

The Imaging procedure shown in the lower part of Fig. 3 is invoked after Clustering to create a page-based image, which is named *PBI* denoted by $\Phi^{(\gamma+1)}$, and to generate a patching script *DFFS* to be disseminated over wireless networks. That is, the Imaging inputs $M\Pi^{(\gamma+1)}$, and a page array of a previous revision $\Phi^{(\gamma)}$, and generates an array of pages that have all sets of resolved functions, i.e., $\Phi^{(\gamma+1)}$ and a patching script that is based on *DC* so that different code can be copied from a source address to a destination address, or inserted or replaced at a source address.

6.1 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.

It has been reasoned that the more complex a function is, the more prone to errors it is, and therefore it is more likely to be modified and to cause more code shift. Also, even a simpler function can have a high influence if it has a large number of references.

Every function that experiences code shift also causes all incoming references to these functions to be updated. In short, the influence of a function depends on the place where the function is located. Consequently, it is critical that we quantify each function's complexity and number of incoming references for the purpose of determining the relative location of the functions.

As an illustration, Figs. 9(a) and (b) show two different images named $\beta'^{(\gamma)}$ and $\beta''^{(\gamma)}$ for the same initial version of the program. The difference is that in $\phi_2^{(\gamma)}$,

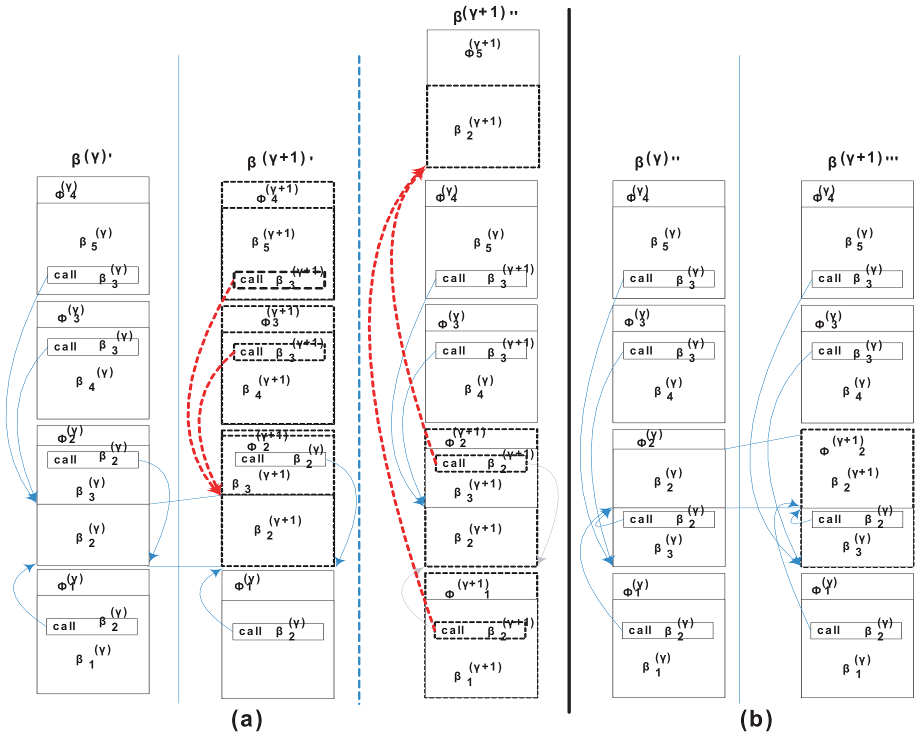


Fig. 9. Different Layouts and Different Updates. (a) In case of the enlarged function, $\beta_2^{(\gamma+1)}$ placed at lower address than $\beta_3^{(\gamma)}$. (b) In case of $\beta_2^{(\gamma+1)}$ placed at higher address than $\beta_3^{(\gamma)}$.

the former arranges the function $\beta_2^{(\gamma)}$ before $\beta_3^{(\gamma)}$ while the latter does $\beta_3^{(\gamma)}$ before $\beta_2^{(\gamma)}$. The point of this example is to show that a good initial ordering even just within $\phi_2^{(\gamma+1)}$ can lead to dramatically lower perturbation to the code memory, when function $\beta_2^{(\gamma+1)}$ is enlarged.

Starting with $\beta'^{(\gamma)}$, Fig. 9(a) may evolve into either $\beta'^{(\gamma)}$ or $\beta''^{(\gamma)}$, depending on how the enlarged function $\beta_2^{(\gamma+1)}$ is kept in the original page ($\phi_2^{(\gamma+1)}$) or put in a newly allocated page ($\phi_5^{(\gamma+1)}$), respectively. If kept in the same page, then $\beta_2^{(\gamma+1)}$ still has the same starting address, and therefore none of its callers need to change, but $\beta_3^{(\gamma+1)}$ is shifted, and therefore all of its callers must be updated, including $\beta_4^{(\gamma+1)}$ in $\phi_3^{(\gamma+1)}$ and $\beta_5^{(\gamma+1)}$ in $\phi_4^{(\gamma+1)}$. In total, three pages must be updated. On the other hand, if the enlarged function $\beta_2^{(\gamma+1)}$ is placed in a newly allocated $\phi_5^{(\gamma+1)}$, callers of $\beta_2^{(\gamma+1)}$ need to be updated, and they also affect three pages, $\{\phi_1^{(\gamma+1)}, \phi_2^{(\gamma+1)}, \phi_5^{(\gamma+1)}\}$, as shown in $\beta''^{(\gamma)}$, but it uses a total of five pages instead of four as $\beta'^{(\gamma)}$ does.

Fig. 9(b) shows that a different initial image ($\beta''^{(\gamma)}$) can reduce the number of affected pages from three down to one, simply by ordering $\beta_3^{(\gamma)}$ before $\beta_2^{(\gamma)}$ on Page 2. The function $\beta_2^{(\gamma)}$ can be enlarged within $\phi_2^{(\gamma)}$ without affecting the starting address of either $\beta_2^{(\gamma+1)}$ or $\beta_3^{(\gamma)}$. Therefore, none of their callers need to be updated, and the only page that needs to be updated is $\phi_2^{(\gamma+1)}$.

How does one determine what functions are more likely to be modified than others? Several software metrics can be considered, and Graves et al classifies fault prediction techniques into *product measures* (syntactic, including code size, degree of statement nesting, number of acyclic execution paths, the in-degree of a statement block, etc.) and *process measures* (change history). Most product measures turned out to be highly correlated to the lines of code, which is not a good predictor of faults [Graves et al. 2000].

However, we have found it effective to adopt a compound metric, called the *influence*, by combining *cyclomatic complexity* and the *number of references* to each function. Cyclomatic complexity [McCabe 1976] is based on the number of linearly independent paths of each function in its control flow graph. The equation for the cyclomatic complexity is as follows:

$$M = E - N + 2P \tag{15}$$

where M is cyclomatic complexity, E is the number of *edges*, N is the number of *vertices*, and P is the number of *connected components* in the graph. Note that these variable names are taken from the original definition and are not related to those defined earlier in this paper. We use the tool called C & C++ Code Counter (CCCC) [Littlefair 2006] to obtain the quantitative value of cyclomatic complexity.

To quantify the influence of a function on other functions, we define the Influence Equation $ie(f_i)$ of the function f_i as the cyclomatic complexity of f_i weighted by the number of references to f_i, as shown below:

$$ie(f_i) = cc(f_i) \times |r^{-1}(f_i)| \tag{16}$$

where $cc(f_i)$ denotes the cyclomatic complexity of the function, and SF is a set of functions $\{f_1, f_2, f_3, ..., f_n\}$ within a page.

Equation (21) 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 the 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 SC, and SR are defined as follows.

$$SC = \{c_1, c_2, c_3, ..., c_n\} \tag{17}$$

$$c_j = cc(f_j) \tag{18}$$

$$SR = \{r_1, r_2, r_3, ..., r_n\} \tag{19}$$

$$r_i = NR(\{f_i\}, SF \setminus \{f_i\}) \tag{20}$$

$$ie(f_i) = \sum_{j=1}^{m} |R(SF \setminus \{f_i\})| \times c_j \tag{21}$$

where 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

```
ORDERING(URPBI)
1  for $k \leftarrow 0$ to $M$ ▷ the number of pages
2        do $Page_k \leftarrow$ GETTINGPAGE($k$)
3           $PageList_k$
              $\leftarrow$ ORDERINGWITHINPAGE($Page_k$)
4           $PBI$
              $\leftarrow$ PAGEBASEDIMAGE($PageList_k$, $URPBI$)
5  return $PBI$
```

(a)

```
ORDERINGWITHINPAGE(SF)
1  $NSF \leftarrow [\,]$
2  $ie_{min} \leftarrow \infty$
3  while $SF \neq \{\}$
4        do for each element $f_k \in SF$
5              do $ie_k \leftarrow ie(f_k)$
6                 if $ie_{\min} > ie_k$
7                    then $ie_{\min} \leftarrow ie_k$
                          $f_{\min} \leftarrow f_k$
8           $SF \leftarrow SF \setminus \{f_{\min}\}$
9           $NSF.enqueue(f_{\min})$
10 return $NSF$
```

(b)

Fig. 10. (a) ORDERING algorithm. (b) ORDERINGWITHINPAGE algorithm.

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.

6.2 Patching Scripts

The IMAGING procedure generates a patching script based on the set of the properties of different image, DC, 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 patching script is similar to the previous work such as [Reijers and Langendoen 2003] in that it includes three primitives: insert, replace, and copy. The insert and replace primitives have the format of a variable-byte opcode and variable-byte length destination address with n bytes of data or instructions. The format of the copy primitive is a variable-byte opcode, variable-byte length of source address, variable-byte length of destination address, and variable-byte length of the data or instruction block copied. Each length can be fixed early on when the host and the nodes first communicate with each other, and new primitives can be applied by using the copy primitive formation.

7. EXPERIMENTAL RESULTS

We evaluate the effectiveness of the algorithms as follows. In Section 7.1, we shows the benefits of ORDERING (part of IMAGING) in minimizing the code differences in successive revisions. In Section 7.2, we evaluate CLUSTERING and IMAGING for NOR flash memory based on page-granular access in terms of energy consumption. The first and second test cases are for RX and TX mode images as shown in Table III, and the third set is for FreeRTOS [Real Time Engineers, Ltd. 2010] images shown in Table IV. Without loss of generality, for the purpose of our experiments, we assume power characteristics of Eco platform [Park and Chou 2006] and NOR flash memory [Atmel Corporation 2008]. The characteristics are in Tables I and II, respectively.

Table I. Power characteristics of the experimental platform.

Param	RF Rx	RF Tx	EEPROM	CPU Active	CPU Powerdown	ADC
Current	10.5 mA	19 mA	5 mA	3 mA	2 μA	0.9 mA

Table II. Flash energy consumption (unit: μJ/byte)

Component	Read	Write	Erase
AT29C010A	0.25	0.48	0.48

Table III. Sizes of versions of the RX and TX images [bytes]

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

Table IV. Sizes of versions of FreeRTOS images [bytes]

Version	0	1	2	3	4	5	6	7	8
Image	23046	23554	21767	23554	23554	23569	23796	23583	23583
Revision	9	10	11	12	13	14	15	16	17
Image	23016	22981	22992	22566	22596	22596	22596	22596	22548
Revision	18	19	20	21	22	23	24	25	26
Image	23051	23051	23051	23043	23393	23393	23393	23393	23382
Revision	27	28	29	30	31	32	33	34	35
Image	24521	24511	24511	24370	24370	24370	24520	24628	24628
Revision	36	37	38	39	40	41	-	-	-
Image	24628	24628	24730	24730	24730	24506	-	-	-

7.1 Ordering Experiments

Table III shows two of our test cases: eight revisions of RX (receive) and eight revisions of TX (transmit). Table IV shows the test case with 41 revisions of FreeRTOS images.

These images are compiled by the Small Device C Compiler (SDCC) [SDCC 2009] targeting Eco, an ultra-compact wireless sensor platform. We use the energy characteristics of NOR flash memory [Atmel Corporation 2008] with different page sizes: 128-byte pages are for the RX and TX images, and 1024-byte pages are for the FreeRTOS images in that the sizes of the FreeRTOS images are around nine times larger than those of the RX and TX images.

Each of the TX and RX images has two different types of image layouts. One has the functions in increasing order of the quantitative rank (Eqn. (21)). The other has the functions in decreasing order of the rank, which is a combination of the cyclomatic complexity and the number of references to each function.

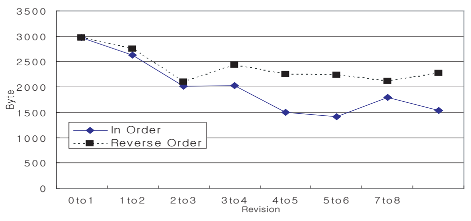
Figs. 11(a) and 11(b) show the code differences in bytes between successive revisions of the RX images and TX images according to the data in Table V. The lines stand for RX and TX images in order of increasing influence, and the dotted lines stand for those ordered in reverse. For the RX and TX test cases, the amount of code difference between successive *in-order* images are consistently smaller than those *reverse-order* images. Not only fewer references need to be modified but also

Table V. Differences between revisions (in bytes): (a) RX (b) TX.

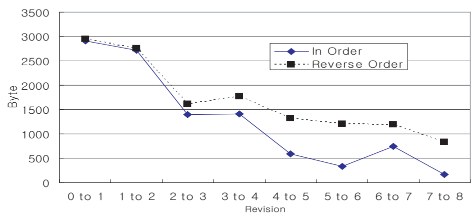
App	Revision	(0 to) 1	2	3	4	5	6	7	8
RX	In Order	2974	2632	2015	2031	1502	1418	1787	1541
	Reverse Order	2974	2752	2105	2436	2242	2238	2117	2271
TX	In Order	2914	2712	1392	1407	595	327	744	167
	Reverse Order	2954	2752	1612	1767	1315	1205	1174	802

Table VI. Similarity between revisions: RX Images; TX Images [%]

App	Revision	(0 to) 1	2	3	4	5	6	7	8
RX	In Order	6.89	13.19	17.59	27.88	47.08	49.64	29.56	44.79
	Reverse Order	6.89	9.23	13.91	13.49	21.00	20.53	16.55	18.63
TX	In Order	6.89	13.19	17.59	27.88	47.08	49.64	29.56	44.79
	Reverse Order	6.89	9.23	13.91	13.49	21.00	20.53	16.55	18.63



(a)



(b)

Fig. 11. Differences between revisions (a) RX images (b) TX images [bytes].

less code needs to be shifted.

Fig. 12(a) shows the benefits of ORDERING for RX images when compared to those ordered in reverse according to the data in Table VI. The total code size of the RX images in order is 17% smaller than that in reverse order. In addition, Fig. 12(b) shows the benefits of ORDERING for the TX images compared to those in reverse order. The total code size of the RX images in order is 24.5% smaller than that of the reverse order, also according to the data in Table VI. In these comparisons, Rsync [Tridgell and Mackerras 1996] is used as the diff algorithm.

For FreeRTOS images, each image has two different types of image layouts. One is in increasing order of the quantitative rank, and the other is in alphabetical order. Of the 41 revisions, 25 revisions actually changed the code; the other revisions changed either comments or formatting without affecting the compiled code.

Fig. 14 (a) shows the code difference between successive revisions of FreeRTOS images, and (b) shows the accumulated code difference in bytes, according to the data in Tables VII and VIII, respectively.

7.2 Clustering and Imaging Experiments

Based on the same cases as Section 7.1, our comparison results consist of (a) our approach (CLUSTERING and IMAGING), (b) fragmented layouts with slop spaces [Koshy and Pandey 2005], and (c) feed back linking [von Platen and Eker 2006] approach to fragmented layouts by placing functions to free spaces to reduce code

Table VII. Similarity of FreeRTOS images[%]

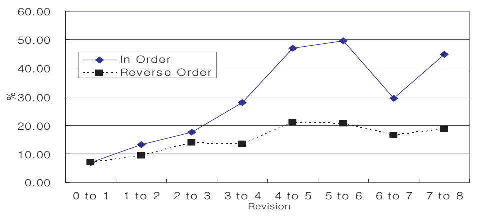
Revision	(0 to) 1	2	3	4	5	6	7	8
In Order	54.86	43.07	39.94	***100***	74.31	73.83	64.54	***100***
Alphabetic	28.85	24.78	21.58	***100***	45.54	44.16	35.11	***100***
Revision	9	10	11	12	13	14	15	16
In Order	11.04	59.53	78.54	69.61	63.03	***100***	***100***	***100***
Alphabetic	10.86	31.85	62.55	28.50	51.80	***100***	***100***	***100***
Revision	17	18	19	20	21	22	23	24
In Order	69.61	63.03	***100***	***100***	82.24	82.77	***100***	***100***
Alphabetic	44.15	63.03	***100***	***100***	55.25	55.24	***100***	***100***
Revision	25	26	27	28	29	30	31	32
In Order	***100***	82.85	27.74	76.87	***100***	85.56	***100***	***100***
Alphabetic	***100***	55.48	27.41	49.86	***100***	55.28	***100***	***100***
Revision	33	34	35	36	37	38	39	40
In Order	68.96	74.63	***100***	***100***	***100***	22908	***100***	***100***
Alphabetic	41.39	47.26	***100***	***100***	***100***	62.87	***100***	***100***
Revision	41	-	-	-	-	-	-	-
In Order	65.41	-	-	-	-	-	-	-
Alphabetic	39.05	-	-	-	-	-	-	-

Table VIII. Code Size Difference of FreeRTOS images[byte]

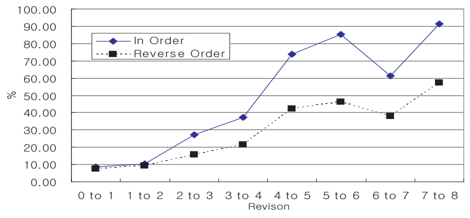
Revision	(0 to) 1	2	3	4	5	6	7	8
In Order	10632	12393	14147	***0***	6055	6227	8363	***0***
Alphabetic	16759	16373	18547	***0***	12835	13287	15303	***0***
Revision	9	10	11	12	13	14	15	16
In Order	20476	9341	1831	10534	4850	***0***	***0***	***0***
Alphabetic	20517	15661	8611	16134	10890	***0***	***0***	***0***
Revision	17	18	19	20	21	22	23	24
In Order	6852	8523	***0***	***0***	4092	4030	***0***	***0***
Alphabetic	12592	13923	***0***	***0***	10312	10470	***0***	***0***
Revision	25	26	27	28	29	30	31	32
In Order	***0***	4009	17719	5670	***0***	3519	***0***	***0***
Alphabetic	***0***	10409	17799	12290	***0***	10899	***0***	***0***
Revision	33	34	35	36	37	38	39	40
In Order	7610	6248	***0***	***0***	***0***	1822	***0***	***0***
Alphabetic	14370	12988	***0***	***0***	***0***	9182	***0***	***0***
Revision	41	-	-	-	-	-	-	-
In Order	8476	-	-	-	-	-	-	-
Alphabetic	14936	-	-	-	-	-	-	-

Table IX. Total Energy Consumption of RX, TX and FreeRTOS Image Updates by comparing with other flash-based image layouts in [μJ]

	Our Approach	Fragmented Layout (Koshy)	Feed-back Linking (von Platen)
RX	2290.02	3709.67	3689.84
TX	1769.23	3165.70	3205.35
FreeRTOS	19708.00	28086.20	28285.90



(a)



(b)

Fig. 12. Similarity between revisions: (a) RX images (b) TX images [%].

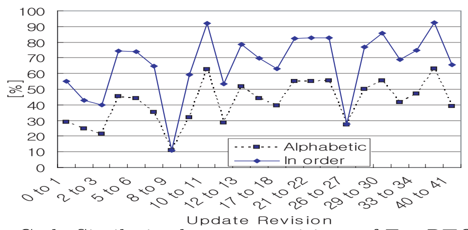
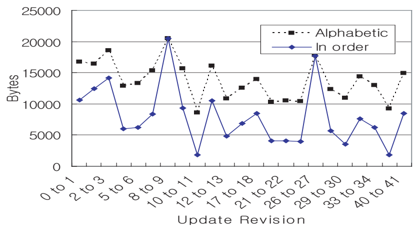
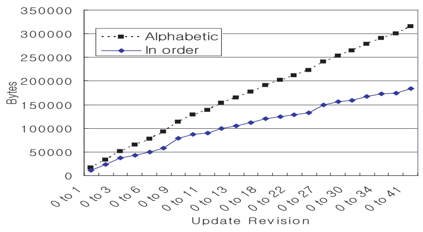


Fig. 13. Code Similarity between revisions of FreeRTOS Images

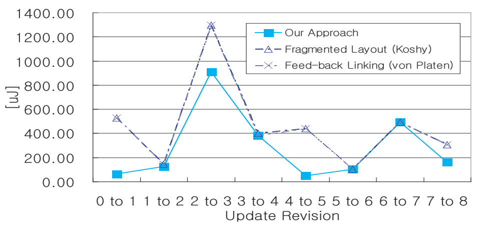


(a)

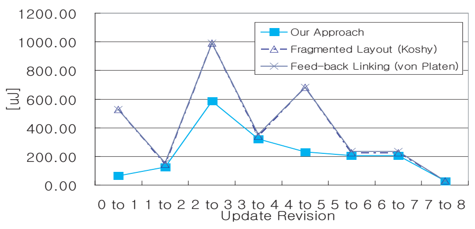


(b)

Fig. 14. (a) Difference and (b) Accumulated Difference of FreeRTOS Images in bytes.



(a)

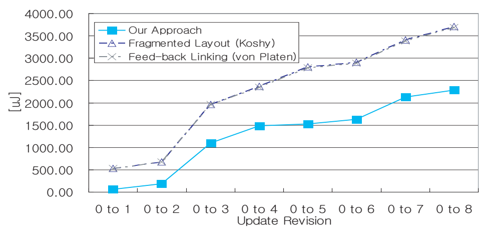


(b)

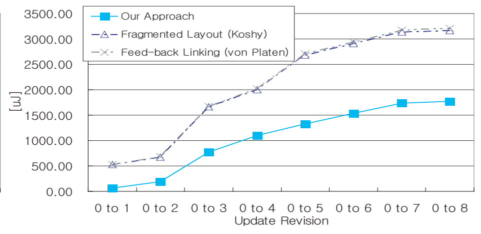
Fig. 15. Energy Consumption by comparing with other flash-based image layouts: (a) RX (b) TX.

Table X. Total number of pages used by different code update schemes

Test Case	Our Approach	Fragmented Layout (Koshy)	Feed-back Linking (von Platen)
RX	271	471	459
TX	271	311	335
FreeRTOS	153	186	191

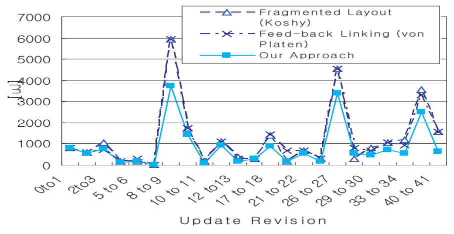


(a)

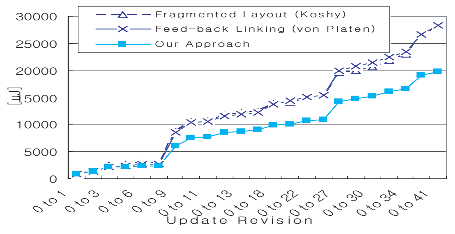


(b)

Fig. 16. Accumulated Energy Consumption of (a) RX Images (b) TX Images by comparing with other flash-based image.



(a)



(b)

Fig. 17. Energy Consumption of FreeRTOS by comparing with other flash-based image: (a) Each update Energy Consumption (b) Accumulated Energy Consumption .

Table XI. Energy breakdown of updating RX, TX, and FreeRTOS images [%]

Process		Code Memory Access			RF Execution		
Scheme	Case	Code relocation	Code difference update	Additional update	Code relocation	Code difference update	Additional update
RPFU	RX	36.46	0.46	0.00	0.01	58.86	4.21
Layout1		58.85	0.32	0.00	0.00	41.50	3.33
Layout2		55.26	0.32	0.00	0.01	40.95	3.46
RPFU	TX	32.54	0.52	0.00	0.00	66.94	3.33
Layout1		49.96	0.37	0.00	0.00	46.83	2.69
Layout2		50.11	0.35	0.00	0.00	44.84	3.07
RPFU	FreeRTOS	42.21	0.39	0.00	0.00	50.97	6.43
Layout1		70.27	0.30	0.00	0.00	24.71	4.72
Layout2		67.37	0.30	0.00	0.00	27.75	4.58

shift incidents. The purpose of the comparison is to show the advantages of our layouts. Table X shows the numbers of pages used by different code update schemes. The numbers of pages for the first versions are not considered, because each scheme has different code layout strategies, and code updates are preformed after the first versions. Our technique saves 41.11% and 41.80% energy for the RX case, 31.4% and 37.9% for TX, 29.83% and 30.33% for FreeRTOS over both other fragmented layouts approaches based on the data in Table IX.

7.3 Breakdown of Energy Consumption

The energy consumption for code updates is broken down into two dominant parts: code memory access and RF execution. Moreover, code updates are performed by a code memory access and RF execution process, and each process is broken down to code relocation, code difference update, and additional update process.

Table XI shows the energy consumption of each code update process. According to Tables I and II, RPFU's portions of the RF execution process are larger than those of Layout 1 (Fragmented Layout) and Layout 2 (Feedback Linking). This means that our RPFU technique effectively reduces much energy for code updates.

7.4 Discussion

One limitation is that our technique does not consider updating (constant) data, such as strings and scalars. Our cost function uses structural analysis to predict likely changes, but changes to data do not always follow such patterns. Another limitation mentioned in the assumption is that each function's image fits in a page, and partitioning is currently done manually, although we believe it can be automated. Our use of cyclomatic complexity falls under the category of *product measure*, namely the syntactic aspect of the program, rather than *process measure*, namely the change history, because we are predicting the future based on the very first version of the program. One possible future extension is to incorporate some process measures so that modification patterns from one project can help better predict the modification patterns of another new project.

8. 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 evolve well by minimizing not only the size of the patching scripts between successive revisions 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. The RPFU program and test cases in this paper have been open-sourced [Kim 2010].

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: Lists of Symbols used in Algorithms and Equations 2 to 8

ClusteringTwoVertices():	clusters the vertex with the parter vertex
CurrentSave:	the number of changes to intra-page references
CurrentVertex:	the vertex RecursiveClustering() is currently visiting
DFFS:	patching script
DSS:	stack for the data structure of modified functions
$E_{\text{buf}}^{(\text{buffer access})}(nBytes)$:	input : buffer access such as read or write and size of bytes output: energy consumption per byte $[J/byte] \times nBytes$ including access overhead energy consumption
$E_{\text{CPU}}^{(\text{process})}(nBytes)$:	input : process and size of bytes output : energy consumption per instruction [J/instruction] $\times nBytes$ for executing a process that is one of FLASH, BUFFER, and RF process
$E_{\text{flash}}^{(\text{flash memory access})}(nBytes)$:	input : flash memory access such as read, write, or erase and size of bytes output: energy consumption per byte $[J/byte] \times nBytes$ including access overhead energy consumption
$E_{\text{RF}}(nBytes)$:	energy consumption per byte $[J/byte] \times nBytes$ for RF including operation overhead energy consumption
EstimatePtrVertex():	works based on FindPartnerVertex to find a partner vertex by measuring the number of inter-page callers. It attempts combining with a caller, independent, or callee.
f_k:	the k^{th} function in *DSS*
MF:	stack for modified functions
NSC:	new source code
NSF:	new sequence of functions
OPBI:	old page-based image denoted by $\Phi^{(\gamma)}$
OS:	old source code
$Page_k$:	the k^{th} page denoted by $\pi^{(\gamma+1)}$
PartnerVertex:	partner vertex (caller, independent vertex, or callee)
PBCG:	page-based call graph
PBI:	page-based image
PreSave:	previous number of changes to intra-page references
PreVertex:	caller or independent vertex. In Fig. 6, a *PreVertex* f_1 is a caller to the *CurrentVertex*, f_7
UpdateCostForEnF():	update the energy cost for an enlarged function
UpdateCostForRmF():	update the energy cost for a removed function
UpdateCostForShF():	update the energy cost for a shrunk function
SF:	unsorted functions
URPBI:	unresolved page-based image denoted by $\Pi^{(\gamma+1)}$
VS:	stack for vertices

Subscripts	
buf:	buffer operation process
erase:	flash memory erasure access
flash:	flash operation process
read:	memory read access
write:	memory write access
RF:	RF communication operation process

REFERENCES

Atmel Corporation. 2008. AT29C010A full data sheet :1-megabit 5-volt only flash memory. `http://www.atmel.com/dyn/resources/prod_documents/doc0394.pdf`.

Graves, T. L., Karr, A. F., Marron, J. S., and Siy, H. 2000. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering 26,* 7 (July), 653–661.

Jeong, J. and Culler, D. 2004. 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)*. IEEE Computer Society, Los Alamitos, CA, USA, 25–33.

Kim, J. 2010. Remote progessive firmware update project. `http://rpfu.sourceforge.net/`.

Koshy, J. and Pandey, R. 2005. Remote incremental linking for energy-efficient reprogramming of sensor networks : Wireless sensor networks. In *Proceedings of the Second European Workshop*. IEEE Press, Piscataway, NJ, USA, 354–365.

Li, W., Zhang, Y., Yang, J., and Zheng, J. 2007. UCC: update-conscious compilation for energy efficiency in wireless sensor networks. In *Proceedings of the 2007 PLDI conference*. Vol. 42. ACM, New York, NY, USA, 383–393.

Littlefair, T. 2006. CCCC – C and C++ Code Counter. `http://cccc.sourceforge.net/`.

Marrón, P. J., Gauger, M., Lachenmann, A., Minder, D., Saukh, O., and Rothermel, K. 2006. FlexCup: A flexible and efficient code update mechanism for sensor networks. In *Wireless Sensor Networks*, K. Römer, H. Karl, and F. Mattern, Eds. Lecture Notes in Computer Science, vol. 3868. Springer, Berlin / Heidelberg, 212–227.

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

Park, C. and Chou, P. H. 2006. Eco: Ultra-wearable and expandable wireless sensor platform. In *Proc. Third International Workshop on Body Sensor Neteworks*. IEEE Computer Society, Los Alamitos, CA, USA, 162–165.

Park, C., Lim, J., Kwon, K., Lee, J., and Min, S. L. 2004. Compiler-assisted demand paging for embedded systems with flash memory. In *Proceedings of the International Conference on Embedded Software (EMSOFT)*. ACM, New York, NY, USA, 114–124.

Real Time Engineers, Ltd. 2010. FreeRTOS free, portable, open source, royalty free, mini real time kernel. `http://www.freertos.org/`.

Reijers, N. and Langendoen, K. 2003. Efficient code distribution in wireless sensor networks. In *Proceedings of the 2nd ACM International Workshop on Wireless Sensor Networks and Applications (WSNA '03)*. ACM, New York, NY, USA, 60–67.

SDCC. 2009. Small Device C Compiler. `http://sdcc.sourceforge.net/`.

Tridgell, A. and Mackerras, P. 1996. The rsync algorithm. Tech. Rep. TR-CS-96-05, Department of Computer Science, The Australian National University. June.

von Platen, C. and Eker, J. 2006. Feedback linking: optimizing object code layout for updates. *SIGPLAN Not. 41,* 7, 2–11.