# Optimizing Relocatable Code for Efficient Software Update in Networked Embedded Systems

WEI DONG, CHUN CHEN, JIAJUN BU, and WEN LIU, Zhejiang University

Recent advances in Microelectronic Mechanical Systems (MEMS) and wireless communication technologies have fostered the rapid development of networked embedded systems like wireless sensor networks. System software for these self-organizing systems often needs to be updated for a variety of reasons. We present a holistic software update (i.e., reprogramming) system called R3 for networked embedded systems. R3 has two salient features. First, the binary differencing algorithm within R3 (R3diff) ensures an optimal result in terms of the delta size under a configurable cost measure. Second, the similarity preserving method within R3 (R3sim) optimizes the binary code format for achieving a large similarity with a small metadata overhead. Overall, R3 achieves the smallest delta size compared with other software update approaches such as Stream, Rsync, RMTD, Zephyr, Hermes, and R2 (e.g., 50%–99% reduction compared to Stream and about 20%–40% reduction compared to R2). R3's implementation on TelosB/TinyOS is lightweight and efficient. We release our code at http://code.google.com/p/r3-dongw.

Categories and Subject Descriptors: C.2.1 [**Computer Communication Networks**]: Network Architecture and Design—*Distributed networks*; D.4.7 [**Operating Systems**]: Organization and Design

General Terms: Design, Experimentation, Performance

Additional Key Words and Phrases: Wireless sensor network, software update, reprogramming, differencing algorithm

**ACM Reference Format:**
Wei Dong, Chun Chen, Jiajun Bu, and Wen Liu. 2014. Optimizing relocatable code for efficient software update in networked embedded systems. ACM Trans. Sensor Netw. 11, 2, Article 22 (July 2014), 34 pages.
DOI: http://dx.doi.org/10.1145/2629479

## 1. INTRODUCTION

Recent advances in Microelectronic Mechanical Systems (MEMS) and wireless communication technologies have fostered the rapid development of networked embedded systems like wireless sensor networks. System software for these self-organizing systems often needs to be updated for a variety of reasons—fixing bugs, changing network functionality, tuning system parameters, and the like. In our recent efforts in deploying a large-scale sensor network system—GreenOrbs during December 2010 through April 2011, we regularly faced the requirement of software upgrade. For example, our software version increased from version 158 to version 285 in the SVN repository. Although

This work is supported by the National Science Foundation of China under Grant No. 61202402, and the Research Fund for the Doctoral Program of Higher Education of China (20120101120179).
Author's address: W. Dong, C. Chen, J. Bu, and W. Liu, College of Computer Science, Zhejiang University, Hangzhou, 310027, P. R. China; emails: {dongw, chenc, bjj}@zju.edu.cn, liuwen@emnets.org.

DOI: http://dx.doi.org/10.1145/2629479

some modifications were intended for good readability and maintenance, we manually identified that at least 15 (i.e., 12%) of the updates were critical to system performance, including fixing link estimation bugs, adding additional metrics for diagnosis, and more.

The current software update technologies are often unsatisfactory. The serial programming approach does not scale well because it requires collecting all nodes back, attaching to computers to "burn" new codes, and redeploying all nodes into the field. Researchers have thus introduced wireless reprogramming techniques to enable each node to automatically update the software through wireless communications. The default wireless reprogramming approach—Deluge [Hui and Culler 2004] for TinyOS, however, has a large impact on the system performance because it often needs to transfer a large binary code. The dissemination of a large binary code will incur a huge amount of transmissions, which is undesirable due to two reasons. First, it is energy-inefficient. Second, it may cause severe interference in the network. It is also worth noticing that a long reprogramming time may also interrupt the normal operation of a sensor network. For some applications, such as structural monitoring or event detection, a long interruption of the normal operation will likely cause the miss of critical system events. Therefore, we usually require the dissemination process to take place in as short a time as possible.

Many efforts have been made to improve the efficiency of wireless reprogramming, such as virtual machines [Levis and Culler 2002; Koshy and Pandey 2005b], modular designs [Dunkels et al. 2004; Han et al. 2005; Dong et al. 2010], compression [Tsiftes et al. 2008], incremental approaches [Jeong and Culler 2004; Panta et al. 2009; Hu et al. 2009; Panta and Bagchi 2009; Dong et al. 2013], network encoding [Hagedorn et al. 2008; Rossi et al. 2008; Dong et al. 2011], and more. In this work, we consider the incremental approach to improve the reprogramming efficiency.

The rationale behind incremental reprogramming is that the change to the software is often much smaller than the entire code in most software change cases. Hence, transferring only the changed part (which is encoded in a delta file) can significantly reduce the amount of transmissions. Regarding the delta file, there are two factors affecting its size.

(1) The differencing algorithm. It should generate a small delta file given two binary files. Existing binary differencing algorithms suffer from two limitations [Jeong and Culler 2004; Hu et al. 2009]. First, they do not ensure an optimal result in terms of the delta file size. Second, some of them may incur a large overhead in terms of execution time and memory consumption. We aim to propose an optimal differencing algorithm in terms of the delta file size, one with good runtime performance.

(2) The code generation mechanism should generate binaries with large similarity (between successive code versions) so that the differencing algorithm can benefit from the achieved similarity and the delta size can be further reduced. Existing methods achieve a limited similarity, and they may cause memory segmentations [Panta et al. 2009; Panta and Bagchi 2009; Dong et al. 2013]. Some of them may even have a large additional metadata overhead. We aim to propose an optimized binary code format that achieves a large similarity without causing memory segmentation or incurring a large metadata overhead.

We build a holistic reprogramming system called R3. R3 has two salient features. First, the binary differencing algorithm within R3 (R3diff) ensures an optimal result in terms of the delta size under a configurable cost measure. The time and space complexity of R3diff is $O(n^3)$ and $O(n)$, respectively. R3diff significantly improves the runtime performance of RMTD proposed in Hu et al. [2009]. Second, the similarity preserving method within R3 (R3sim) optimizes the binary code format for achieving a large similarity with a small metadata overhead. Overall, R3 achieves the smallest

delta size compared with other incremental approaches such as Rsync [Jeong and Culler 2004], RMTD [Hu et al. 2009], Zephyr [Panta et al. 2009], Hermes [Panta and Bagchi 2009], and R2 [Dong et al. 2013]. R3's implementation on TelosB/TinyOS is lightweight and efficient.

We carefully examine a number of code features such as the number of reference instructions, the number of referenced symbols, and the like. This helps us understand and predict the performance of various incremental approaches. We also compare incremental encoding with general compression. Although delta files are less compressive than native code, the combination of incremental encoding and general compression can further reduce the code size. We release our code at http://code.google.com/p/r3-dongw.

The contributions of this work are summarized as follows.

—We propose an optimized relocatable code format for preserving code similarity with a small metadata overhead.
—We propose an optimal and efficient byte-level differencing algorithm for comparing two binary files. We formally prove the optimality of the proposed algorithm.
—We quantitatively analyze the performance of existing similarity preserving methods including Zephyr [Panta et al. 2009], Hermes [Panta and Bagchi 2009], R2 [Dong et al. 2013], and R3.
—We conduct extensive evaluations. We propose an empirical model to analyze different factors impacting the reprogramming energy consumption. We also analyze the energy improvement by combining incremental reprogramming and compression.

Compared with the conference version of this article [Dong et al. 2013], this journal version contains the following important extensions.

—We add Section 4 to better motivate our work.
—We present more detailed description on R3sim and R3diff in Section 5. Also, we add descriptions on the code dissemination and reconstruction process in Section 5.
—We add Section 6 to analytically compare R3 with existing approaches.
—We present a much more detailed evaluation on R3 in Section 7.
—We add Section 8 to discuss several important issues about R3.

The rest of this article is structured as follows. Section 2 describes the background. Section 3 introduces the related work. Section 4 describes the motivation behind the work. Section 5 presents the design details. Section 6 analytically compares R3 with existing approaches. Section 7 shows the evaluation results. Section 8 discusses several important issues about R3. Finally, Section 9 concludes the paper and gives future research directions.

## 2. BACKGROUND

This section provides necessary background on wireless reprogramming.

### 2.1. Loadable Binary File Format

In wireless programming approaches, sensor nodes will store a loadable binary file in the external flash after reception. The loadable binary file will be loaded onto the program flash for execution by a loader.

The binary file format depends on the loader implementation on each node. If each node has an ELF [TIS Committee 1995] loader, the ELF file format can be used. TinyOS/Deluge [Hui and Culler 2004] uses a simple binary file format for transferring and loading. At a high level, the file consists of several sections. Each section is described by a 4-byte memory starting address, a 4-byte section length, followed by the binary data of the specified length. We refer to this file format as .raw. We also use this file format in R3.

The .raw file can be interpreted by the default bootloader in TinyOS/Deluge—tosboot. Tosboot loads the .raw file from the external flash to the specified program memory address.

### 2.2. Incremental Reprogramming

The key idea of incremental reprogramming is to transfer only the delta between the old code and the new code. Typically, the delta file size is much smaller than the entire new code size. Upon reception of the delta file, each node reconstructs the new code by patching the delta to the old code.

The delta file format usually encodes two kinds of commands. The ADD command inserts a byte sequence of a specified length. The COPY command copies a byte sequence from the old code to the new code. These two commands consume certain bytes to make them interpretable. Specifically, we use commands of the following forms (the subscript denotes the number of bytes consumed by each field):

$\texttt{ADD}_1$ $\texttt{<n>}_2$ $\texttt{<BYTE1..BYTEn>}_n$,

where `n` is the number of added bytes. We use $\alpha = 3$ to denote the number of bytes consumed by the first two fields in the ADD command. This command costs a total of $\alpha + n = 3 + n$ bytes.

$\texttt{COPY}_1$ $\texttt{<n>}_2$ $\texttt{<old\_addr>}_2$,

where `n` is the length of copied bytes, and `old_addr` is the start address in the old code. We use $\beta = 5$ to denote the number of bytes consumed by the COPY command. This command costs a total of $\beta = 5$ bytes. It is worth noting that the start address in the new code is not needed because the new code is reconstructed in sequential order.

To minimize the delta size, the differencing algorithm should take this encoding overhead into account.

### 2.3. Relocatable Code

Relocatable code is produced by the compiler, and all memory references in the code needing relocation are specially marked and can be relocated by the linker. Dynamic relocatable code can be relocated during load-time linking.

It is worth mentioning the concept of Position Independent Code (PIC). PIC is a term for code that can run in any address space. This differs from relocatable code, which requires special processing by a link editor or program loader to make it suitable for execution at a given location. PIC uses other mechanisms, such as PC relative addressing instead of absolute addressing. PIC can effectively reduce the metadata overhead compared to relocatable code. Another benefit of PIC for PC software is that the code can be shared (and loaded to different virtual memory addresses) among different processes. The same benefit does not exist on sensor nodes without MMUs.

However, PIC needs compiler support. For example, on the MSP platform, no compiler is known to fully support PIC [Dunkels et al. 2006].

## 3. RELATED WORKS

In this section, we briefly discuss various works in the field of wireless reprogramming.

### 3.1. Full Image Replacement

Deluge [Hui and Culler 2004] provides a complete reprogramming support for TinyOS applications, including the Deluge protocol for code dissemination, the loading and rebooting mechanisms for code execution. Because sensor nodes need to be

reprogrammable multiple times, Deluge transfers the full image of the application along with the reprogramming protocol.

Stream [Panta et al. 2007] reduces the transferred code size by preinstalling the reprogramming protocol on the external flash as another application image (i.e., the reprogramming image) such that the reprogramming protocol does not need to be transferred during dissemination. However, Stream still needs to transfer the full application image.

### 3.2. Loadable Modules

SOS [Han et al. 2005], Contiki OS [Dunkels et al. 2004, 2006], and FlexCup [Marrón et al. 2006] support loadable modules. In these systems, individual modules can be loaded dynamically on the nodes. Specific challenges exist in these systems. First, they require disseminating symbol tables and relocation tables for linking and relocating. These may be quite large, typically 45% to 55% of the object file [Panta and Bagchi 2009]. Second, they make extensive use of the program flash.

Elon [Dong et al. 2010] addresses this issue by introducing the concept of replaceable component. In the network reprogramming process, only the replaceable component needs to be disseminated, thus greatly reducing the dissemination cost. Compared to other modular OSes such as Contiki OS and SOS, which enable reprogramming on a modular basis, Elon does not incur the overhead of relocation entries.

### 3.3. Compression

Sadler and Martonosi [2006] investigate the design issues involved in adapting compression algorithms specifically geared for sensor nodes. Tsiftes et al. [2008] implement the gzip algorithm on sensor nodes for reducing the size of executable modules.

### 3.4. Virtual Machines

Several systems, such as Maté [Levis and Culler 2002] and VM* [Koshy and Pandey 2005b], provide virtual machines that run on resource-constrained sensor nodes. They allow disseminating a small code size because the virtual machine code is much more compact than the native code. However, the virtual machine code is less expressive than the native code since virtual machine code is compiled or translated to native code. VM code trades off, to different degrees, less flexibility in the kinds of tasks that can be accomplished [Panta and Bagchi 2009]. For example, VM code for sensor networks usually cannot perform low-level I/Os such as operations on registers.

### 3.5. Incremental Approaches

Incremental reprogramming [Jeong and Culler 2004; Panta et al. 2009; Hu et al. 2009; Panta and Bagchi 2009; Dong et al. 2013] is a technique to reduce the transferred code size by disseminating the delta file.

Jeong and Culler [2004] modify the Rsync algorithm for efficient incremental reprogramming for sensor nodes. Such a block-based algorithm, suitable for handling large files in computers, is inappropriate for energy-efficient reprogramming for micro-embedded systems. The RMTD algorithm proposed in Hu et al. [2009] is a byte-level differencing algorithm with $O(n^3)$ time complexity and $O(n^2)$ space complexity, where $n$ denotes the file size (in bytes) for comparison. It only ensures optimal results under $\alpha = 0$ (see definition of $\alpha$ in Section 2.2). In practice, the large space requirement makes it unsuitable to scale to increasingly complex software programs. For example, on a PC (with CPU Quad 2.66GHz, RAM 4GB), the memory consumption increases to 1.3GB when comparing files with 100KB, and increases to approximately 5GB when comparing files with 200KB. For more complex programs (e.g., programs for iMote2

nodes that can accommodate a maximum of 256MB program code), it is expected that the huge memory consumption will prohibit its use on current PCs.

Koshy and Pandey [2005a] mitigate the effects of function shifts by using slop regions after each function in a program so that the function addresses do not change when each function grows within the slop region. Such an approach, unfortunately, leads to fragmentation and inefficient use of the program flash. Besides, it only handles growth of functions up to the slop region boundary.

Zephyr [Panta et al. 2009] mitigates the effects of function shifts by using a function indirection table. All function calls in the program are indirected via fixed jump table slots to their actual implementations. The approach does not handle instructions other than call, which can also contain references to functions (e.g., br and mov for TelosB nodes). Hermes [Panta and Bagchi 2009] mitigates the effects of data shifts by first pinning variables to the same locations by source-level modifications and then adopting two different approaches for handling data shifts. For the Von Neumann-based TelosB nodes, it places uninitialized variables (in the .bss section) in the program flash instead of RAM, in order to preserve similarity in the case of data shifts. Such an approach leads to decreased execution efficiency because each access to the uninitialized variables will incur flash I/O operations that are more costly than RAM operations. The concrete number of flash I/Os depends on how many times the program accesses the uninitialized variables. More importantly, the flash memory found on TelosB nodes is designed to withstand approximately 100,000 write/erase cycles on a given sector before it becomes unreliable [Polastre et al. 2005], making the approach unsuitable for memory-intensive and long-term applications. Although both Hermes and R3 use program flash in the loading process, R3 accesses all variables in RAM. For the Harvard-based MicaZ nodes, Hermes leaves a slop region (of 10 bytes) in between the .data and .bss sections, inevitably resulting in fragmentation and inefficient use of the RAM. Also, it only handles growth of .data variables up to the slop region boundary.

R2 [Dong et al. 2013] unifies the approaches for handling function shifts and data shifts by using relocatable code. Memory addresses in the reference instructions are filled with zeros for achieving large similarity between the old and the new code. However, as illustrated in the R2 paper [Dong et al. 2013], the metadata occupies a large portion for complex software and needs to be carefully compressed in a customized manner.

It is worth noting that incremental approaches (including our proposed R3) can work over several update iterations. For example, the current program code version is 1 for all network nodes. We would like to update the program to version 2. Therefore, we compile app v2 at the PC and generate a delta $\Delta_{1,2}$ between app v2 and app v1. After reprogramming, the network nodes have received $\Delta_{1,2}$, reassembling app v2 using app v1 and $\Delta_{1,2}$. Afterward, all nodes in the network are executing app v2. Sometime later, we would like to update the program to v3. We compile app v3 at the PC and generate $\Delta_{2,3}$ between app v3 and app v2. After reprogramming, the network nodes have received $\Delta_{2,3}$, reassembling app v3 using app v2 and $\Delta_{2,3}$. We can see that incremental reprogramming approaches can work over several update iterations.

### 3.6. Dissemination Protocols

We can divide the current code dissemination protocols from two perspectives.

(1) Structured or structureless: Structured protocols mainly include Sprinkler [Naik et al. 2005], CORD [Huang and Setia 2008], and the like. Sprinkler is a reliable bulk data dissemination protocol that constructs a Connected Dominating Set (CDS) with the location information. In addition, Sprinkler uses TDMA to schedule links and enable packet-level pipelining. CORD is another structured dissemination protocol that constructs a CDS and uses coordinated scheduling to enable page-level pipelining and

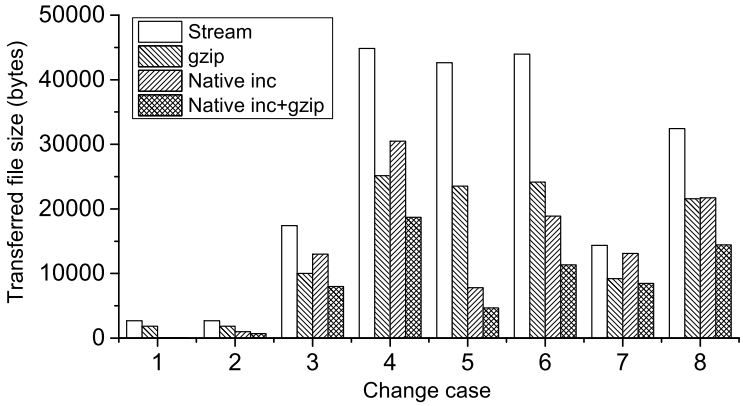


Fig. 1. Comparison among Stream, Stream+gzip, native inc, and native inc+gzip.

sleep control. Structureless protocols mainly include Deluge [Hui and Culler 2004], MNP [Kulkarni and Wang 2005], and ECD [Dong et al. 2011]. Deluge [Hui and Culler 2004] uses a three-way handshake and NACK-based protocol for reliability and employs segmentation (into pages) and pipelining for spatial multiplexing. MNP proposes sender selection to address the concurrent senders problem. ECD improves the sender selection algorithm by considering link quality.

(2) Noncoding or coding-based: Traditional approaches (including the above-mentioned protocols) transmit native packets. In recent years, several coding-based dissemination protocols in WSNs have been proposed, such as Rateless Deluge [Hagedorn et al. 2008], Synapse [Rossi et al. 2008], and ReXOR [Dong et al. 2011]. These protocols use various coding schemes to reduce the number of transmissions. In all these protocols, an encoded packet can be decoded at multiple receivers once a sufficient number of encoded packets are received. Hence, the number of retransmissions can be greatly reduced, and the performance can be significantly improved in lossy networks. On the other hand, the decoding delay can also bring a relatively long dissemination time, especially when the link qualities are excellent and the number of required retransmissions are inherently small. MT-Deluge [Gao et al. 2013] exploits concurrency of sensor nodes (i.e., using two threads to separate the coding and ratio operations), resulting in much improved performance for coding-based dissemination protocols.

Code dissemination protocols are orthogonal to our incremental approach. A better code dissemination protocol will shorten the process to disseminate the delta, resulting in shorter reprogramming time.

## 4. MOTIVATION

This section describes the motivation behind our work.

### 4.1. Motivation for Incremental Reprogramming

The gzip tool, which is an implementation of the LZ77 algorithm, is able to compress large files with good performance. We compare the transferred file size of Stream [Panta et al. 2007], Stream+gzip, and the native incremental approach (using R3diff without similarity preserving method) in Figure 1 using eight software change cases described in Section 7. We can see several facts from the figure.

—In four of the eight cases, the native incremental approach outperforms gzip. This is because the software changes are subject to a small portion of the entire software.

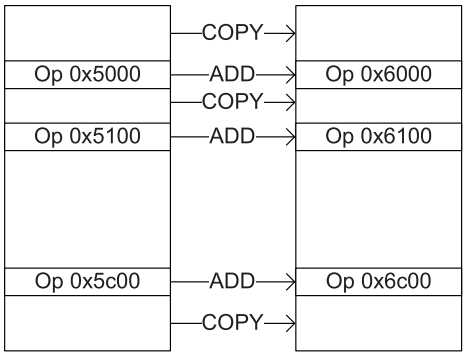


Fig. 2. An example of software change: the new code (right) shifts by 0x1000.

—Indeed, in the other four change cases, the native incremental approach performs worse than gzip. These change cases represent a very large change to the software. For instance, in change case 4, we delete two core components in the program and add one core components in the program.
—Using the incremental approach does not prohibit the use of gzip compression. Native inc+compression performs consistently better than compression.

Note that we use the native incremental approach for illustrative purpose. We can achieve much better performance with similarity preserving methods (e.g., R3sim). We will conduct a deep examination in Section 7.

### 4.2. Motivation for Preserving Code Similarity

Let's consider an example shown in Figure 2. Consider two code segments with $n$ reference instructions. The native incremental approach requires (n + 1) COPY commands and n ADD commands. Assume the address field in the reference instructions consumes 2 bytes, the delta size is thus $(n+1) \times \beta + n \times (\alpha + 2) = 10n + 5$ (see definitions of $\alpha$ and $\beta$ in Section 2.2).

If we can preserve similarity between the two code segments (e.g., the $n$ reference instructions are also the same), only one copy command is required, resulting in significantly smaller delta size. However, there would be metadata overhead since the address fields in the reference instructions should be resolved to the correct ones after code reconstruction.

Although R2 achieves large similarity in the program code, it has a relatively large overhead in the relocation entries (as illustrated in the R2 paper [Dong et al. 2013]). This motivates us to devise an optimized code format that can preserve as large a code similarity as R2 while it has a low metadata overhead.

### 4.3. Motivation for Another Differencing Algorithm

For clarification, we use $\alpha$ and $\beta$ to denote the actual encoding cost in the ADD and COPY commands, and we use $\alpha'$ and $\beta'$ to denote the algorithm's parameters (i.e., algorithm's view of $\alpha$ and $\beta$). In principle, $\alpha'$, $\beta'$ and $\alpha$, $\beta$ should be the same.

We consider the RMTD algorithm [Hu et al. 2009]. To our knowledge, it is the latest differencing algorithm for sensor network reprogramming that achieves better performance than earlier algorithms like Rsync [Jeong and Culler 2004]. RMTD only exposes $\beta'$ as a configurable parameter, whereas $\alpha'$ always equals to 0 regardless of $\alpha$. Can RMTD still generate the optimal command sequence by exposing the $\beta'$ parameter? To answer this question, we give two examples. We consider $\alpha = 3$, $\beta = 5$.

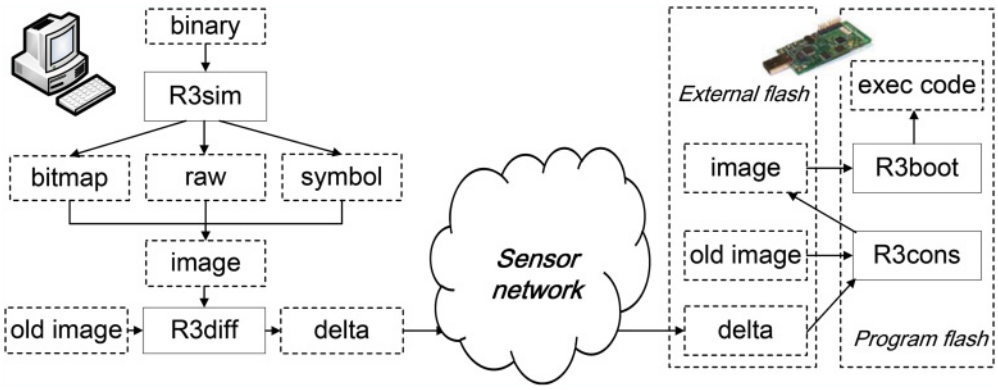


Fig. 3. An overview of the R3 reprogramming system.

(1) Consider two code segments where New = (A B C) and Old = (A B C). With $\beta' = 5$, RMTD will not copy the three bytes because the segment length is smaller than 5 bytes, and RMTD considers that adding them is more beneficial (i.e., the adding cost of 3 bytes is smaller than the copy cost of 5 bytes in the perspective of RMTD). Actually, adding all three bytes consumes $\alpha + 3 = 6$ bytes while copying the three bytes requires $\beta = 5$ bytes.

RMTD does not give the optimal result in this example because it does not copy small segments. Actually, copying common segments with length larger than $\beta - \alpha = 2$ bytes will be beneficial. Hence, we set $\beta' = 2$ in RMTD so that RMTD can copy common segments with length larger than 2 bytes. Now RMTD correctly generates one copy command in this particular example.

(2) Consider another two code segments where New = (A B C D) and Old = (A B C). The RMTD algorithm now generates one copy command and one add command (the cost is $\beta' + 1 = 2 + 1 = 3$ bytes in RMTD's perspective). This translates to the actual encoding cost of $\beta + \alpha + 1 = 9$ bytes. The minimum number of bytes, however, is $\alpha + 4 = 7$ bytes with all four bytes added.

In general, the RMTD algorithm cannot be easily extended to the case when $\alpha > 0$. We use Opt(byte sequence) to denote the minimum number of bytes to construct the byte sequence. In RMTD, when we know Opt(A B C), Opt(A B C D) = Opt(A B C) + 1 since D needs to be added when $\alpha = 0$. When $\alpha > 0$, not only the value of Opt(A B C) can affect the value of Opt(A B C D), but also the actions performed on C. If C is added, then Opt(A B C D) = Opt(A B C) + 1. If C is copied, then Opt(A B C D) = Opt(A B C) + $\alpha$ + 1.

RMTD also incurs a large overhead in terms of execution time and memory consumption, which could be significantly improved. Hence, new algorithms should be devised to ensure the optimal result with a good runtime performance.

## 5. DESIGN

This section introduces R3's design. Figure 3 gives an overview of R3. R3's similarity preserving method, R3sim, augments the GCC compiler by generating code binaries with large similarity. R3's differencing algorithm, R3diff, takes the generated binary files as inputs, and generates a delta for dissemination. The dissemination protocol disseminates delta to all nodes. Upon reception of the delta file, the R3cons component on each node reconstructs the new binary code by patching the delta to the old binary code. Finally, R3's bootloader residing on each node, R3boot, loads the new binary code to the program flash and starts executing the new code.

Currently, we implement R3 for the TelosB platform with the msp430f1611 microcontroller (16bits, 8MHz, 10KB RAM, 48KB program flash, and 1MB external flash). We run the differencing algorithm on a PC with CPU Quad 2.66GHz, 4GB RAM. For the network stack, we require a small data dissemination protocol (e.g., Drip [Tolle and Culler 2005]) as well as a code dissemination protocol (e.g., Deluge [Hui and Culler 2004]). We compile sensor node programs using msp430-gcc version 3.2.3. Application code should wire to a reprogramming support component that includes the Drip dissemination protocol for reliably disseminating wireless commands. We use an optimized Deluge protocol to reliably disseminate the delta. Like other incremental reprogramming approaches, R3 works over several update iterations.

In the following subsections, we discuss individual aspects of R3.

### 5.1. Similarity Preserving Method: R3sim

The key idea of preserving similarity between the old and new code is to make the references to the same symbol unchanged despite code shifts.

R2 achieves this goal by filling the address fields in the reference instructions to zeros. Additional relocation entries are used to resolve the address fields to the correct target addresses. R2 reserves a relocation entry (offset, addr) for each reference instruction. The 2-byte offset field indicates which memory address needs relocation, whereas the 2-byte addr field indicates the actual target address. There are two inefficiencies in R2's relocation table. First, each relocation entry contains an additional word (offset) in order to locate the memory address that needs relocation. Second, the number of relocation entries is proportional to the number of reference instructions, instead of the number of unique symbols.

There does exist a technique called Chained Reference (CR) [Levine 2000] to encode the relocation entries in a more compact manner. The key idea of CR is trying to merge the relocation entries for the same symbol because references to the same symbol in different locations of the program must point to the same target address. To achieve this goal, CR uses references to the same symbol in the program (which contain useless addresses before relocation) to create a linked list. Each relocation entry in the relocation table requires two values: an index to the symbol and a pointer to the first reference of that symbol (i.e., the header of the linked list). The loader performs relocation by traversing each linked list. Using this technique, the size of the relocation table is proportional to the number of unique symbols, instead of to the number of reference instructions.

The CR technique still suffers from two problems. First, each relocation entry still contains 2-byte offset for locating the first memory address that needs relocation. Second, and more importantly, CR needs to fill the target addresses to appropriate pointers, which can decrease the code similarity.

We try to use another technique in R3sim. First, to make references to the same symbol unchanged, we must fill the corresponding addresses to the same value. In R2, all addresses are filled with zeros, whereas in R3 we make more efficient use of the 2 bytes (i.e., they are filled with the symbol indices). A symbol index refers to a symbol entry that contains the actual symbol address. The same symbol has the same index in the old and new code. Using this technique, we only need to include the target addresses for all unique symbols. Moreover, unlike CR, R3 does not decrease code similarity.

Second, we do not use the 2-byte offset field to explicitly indicate which memory address needs relocation. Instead, we use a bitmap indicating which address needs relocation. On the MSP platform, relocation is performed on a 2-byte granularity with only one relocation type. The bit value of 1 indicates that the corresponding memory address at the 2-byte boundary needs relocation, whereas the value of 0 indicates that no relocation is required. This technique eliminates the overhead of the offset field

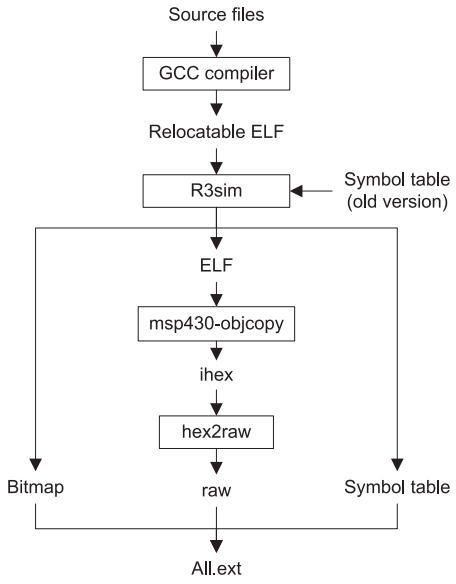


Fig. 4. R3sim's workflow.

while incurring the bitmap overhead of, at most, $C/16 \approx 6.2\%C$, where C is the new code size. Because the bitmap is also transferred by a delta, the actual transferred overhead is even smaller.

Figure 4 shows the workflow of R3's similarity preserving method, R3sim. First, the GCC compiler compiles the source code to a relocatable ELF. R3sim takes inputs of the relocatable ELF and the symbol table of the old version to generate another ELF with similarity preserved. The old symbol table serves as a reference for generating the new symbol table. R3sim will try to allocate the same index for the same symbol. When a symbol is deleted in the new version, R3 marks the entry as empty. When a symbol is added in the new version, R3 will try to use an empty slot before it allocates a new one. R3sim also generates a bitmap file and the new symbol table in binary forms. The generated ELF file is used for extracting the binary instructions. All three files are combined to a single binary file that can be loaded for execution by R3boot.

We summarize R3's benefits over prior approaches:

—R3 keeps the merits of R2. R3 unifies the approach for handling function shifts and data shifts. It also nicely handles the shift of interrupt handlers. R3 preserves as large a code similarity as R2 while effectively reducing R2's metadata overhead.
—R3 does not cause memory segmentation and makes efficient use of memory. While the indirection table of Zephyr requires additional program flash space, R3's symbol table does not occupy program flash space since the symbol table is not loaded onto the program flash.
—R3 does not require the source-level modification used by Hermes. Instead, R3 operates at the binary level. Hermes requires accessing source files to keep similarity of data variables. However, most library files are provided in the form of relocatable binary instead of source files. R3 can be nicely integrated into this compilation model.

### 5.2. Differencing Algorithm: R3diff

R3diff takes the binary files of two successive versions and generates a delta for dissemination. There are some design decisions for R3diff.

(1) R3diff should compare files at the smallest granularity of change. Because most file changes are at the byte level, we devise R3diff to be byte-level. Comparing files at smaller granularity (e.g., bit) will incur larger overhead because the storage requirement of bit-address will be larger than that of byte-address.

(2) However, the bitmap file generated by R3sim changes at the bit level. In this particular case, R3diff performs less effectively without change. Although we can devise an additional bit-level comparison algorithm, we would like to reuse the code of byte-level R3diff. We keep the R3diff algorithm unchanged by providing it extended bitmap files (i.e., 1 bit in the original bitmap files is extended to 1 byte). Correspondingly, we need to set $\alpha' = 8 \times \alpha$, $\beta' = 8 \times \beta$ (see definitions of symbols in Sections 2.2 and 4.3) in order to correctly reflect the encoding costs in bits. We then use an additional utility to compact the generated delta back to bit formats.

We now describe R3diff, the byte-level comparison algorithm that is optimal in terms of the delta size.

We define $opt_i$ as the minimum transferred delta size to construct the first $i$ bytes of the new code. We can get (1) $opt_0 = 0$. (2) $opt_i \geq opt_{i-1}$, where $i \geq 1$. This inequality holds because if we can use $opt_i$ bytes to construct $i$ bytes, we can also use these $opt_i$ bytes to construct $i - 1$ bytes. Hence, the smallest delta size used to construct $i - 1$ bytes should be no larger than $opt_i$.

Figure 5 shows the R3diff algorithm. R3diff's major steps are described as follows.

(1) Generate footprints. We generate footprints (i.e., hash values) for every $p = \beta - \alpha + 1 = 3$ continuous bytes in the old code. The hash values are used to identify Common Segments (CS) in two code versions. Identifying smaller CS is unnecessary because copying a small number of bytes is not more beneficial than adding them. For each hash location, we store an entry representing a length-p segment at a specified offset in the old code. For segments hashed to the same value, we use a linked list to store all entries. The storage requirement is thus (nold $- p + 1$)×sizeof(entry), where nold is the size of the old code.

(2) We compute $opt_i$ for $1 \leq i \leq$ nnew, where nnew is the size of the new code. Here, $i$ denotes the number of constructed bytes in the new code. Hence, $i - 1$ denotes the index of the last constructed byte.

(2.1) To compute $opt_i$, we perform an operation, findk(), on the last index (i-1). The findk function takes an index in the new code as input and returns the smallest index in the new code, denoted as k, such that [k,i-1] matches a segment in the old code.

(2.2) To compute $opt_i$, we introduce two additional notations, $opt_i^A$ and $opt_i^C$. The former represents the minimum needed bytes to construct i bytes with ADD as the last command. The latter represents the minimum number of needed bytes to construct i bytes with COPY as the last command, or, equals to `LARGE_INTEGER` if the last byte cannot be copied (i.e., must be added). By definition, $opt_i = \min(opt_i^A, opt_i^C)$.

(2.3) Compute $opt_i^A$ and $opt_i^C$ as follows.

$$opt_i^A = \min\left(opt_{i-1}^A + 1, opt_{i-1}^C + \alpha + 1\right) \tag{1}$$

$$opt_i^C = \begin{cases} \texttt{LARGE\_INTEGER}, & \text{if } k > i - 1 \\ opt_k + \beta, & \text{otherwise.} \end{cases} \tag{2}$$

Note that the last byte cannot be copied when k > i-1; that is, the findk function cannot find a valid k such that [k,i-1] matches a segment in the old code.

Figure 5 shows the details of the r3diff algorithm. We denote n = max(nold, nnew). The worst-case time complexity of the algorithm is $O(n^3)$, and the average time complexity of the algorithm is $O(n^2)$. The space complexity is $O(n)$.

```
R3diff
Inputs: α, β
p = β − α + 1

R3diff() {
   Generatefootprint();
   Opt[0] = OptA[0] = OptC[0] = 0;
   for (i=1; i<=nsize; i++) {
      k = findk(i-1);
      OptA[i] = min(OptA[i-1]+1, OptC[i-1]+ α+1);
      if (k>i-1) OptC[i] = LARGE_INTEGER;
      else OptC[i] = Opt[k]+β;
      Opt[i] = min(OptA[i], OptC[i]);
   }
}

Generatefootprint() {
   for (i=0; i<=osize-p; i++) {
      val = hash(ofile, i);
      newentry = malloc(sizeof(htentry_t));
      newentry->offset = i; newentry->next = NULL;
      if (hthead[val] == NULL) {
         hthead[val] = httail[val] = newentry;
      } else {
         httail[val]->next = newentry; httail[val] = newentry;
      }
   }
}

int findk(int inew) {
   k = inew+1;
   for (i=inew-p+1; i<=inew && i<=nsize-p; i++) {
      val = hash(nfile, i);
      if (hthead[val]) {
         for (e=hthead[val]; e; e=e->next) {
            ii = i+p-1; jj = e->offset+p-1;
            for ( ; nfile[ii]==ofile[jj] && ii>=0 && jj>=0; i--, j--) ;
            if (ii+1<k) k=ii+1;
         }
      }
   }
   return k;
}
```

Fig. 5. The r3diff algorithm.

We now prove that this algorithm generates the smallest delta size for the given $\alpha$ and $\beta$.

THEOREM 1. *The R3diff algorithm generates the smallest delta size under a given encoding cost of $\alpha$ and $\beta$.*

PROOF. We show the proof by induction.

(1) Basic step. It is easy to see that $opt_0^A = 0$ and $opt_0^C = 0$ are optimal.

(2) Induction hypothesis step. We suppose that $opt_j^A$, $opt_j^C$, and $opt_j$ are optimal for $0 \leq j \leq i-1$.

(3) Induction step. We need to show that $opt_i^A$ and $opt_i^C$ are optimal under this supposition.

We first show $opt_i^A$ is optimal. The previous byte (at index i-2) is either added or copied. If the previous byte is added, $opt_i^A = opt_{i-1}^A + 1$. If the previous byte is copied, $opt_i^A = opt_{i-1}^A + \alpha + 1$. Because $opt_i^A$ denotes the minimum cost with the last byte (at index i-1) added, $opt_i^A = \min(opt_{i-1}^A + 1, opt_{i-1}^C + \alpha + 1)$. Because $opt_{i-1}^A$ and $opt_{i-1}^C$ are optimal, $opt_i^A$ is optimal.

We then show $opt_i^C$ is optimal. When the last byte can be copied, $opt_i^C = opt_k + \beta$, where k is smallest index with which [k,i-1] is a common segment. $opt_i^C$ is optimal because (i) for any fixed k, $opt_k$ is optimal; and (ii) for any k′>k and [k′,i-1] is also a common segment, $opt_{k'} \geq opt_k$. Hence, $opt_i^C = opt_k + \beta$ is optimal. When the last byte cannot be copied, we set $opt_i^C$ to `LARGE_INTEGER` so that the copy action will not be performed on the last byte. □

### 5.3. Dissemination

Our delta dissemination protocol is based on Deluge [Hui and Culler 2004], which works as follows. A node advertises the code version using a Trickle timer [Levis et al. 2004]: the advertisement interval decreases to the minimum when a new code version is found, whereas it doubles to the maximum when the network is steady. When the advertisement of a new code is received, a node sends a request to the sender and the sender will broadcast the new code. Deluge fragments the code into multiple fixed-size pages (e.g., defaults to 1,104 bytes) to enable pipelining, and it employs NACK to achieve reliability.

To make the dissemination protocol better suited to R3, we perform three optimizations.

(1) The original Deluge protocol needs to be wired into the application so that each node can always respond to reprogramming commands and the nodes can be reprogrammable for multiple times. We find that, however, the Deluge code in TinyOS 2.1.1 consumes more than 30KB. It cannot be wired into the GreenOrbs application because the GreenOrbs program already consumes approximately 45KB, approaching the maximum allowable program size of 48KB on TelosB nodes.

Hence, we use the Stream [Panta et al. 2007] optimization for Deluge. Stream installs the Deluge reprogramming protocol on external flash. When reprogramming is required, a node switches to the reprogramming state for receiving the new code. After receiving the new code, the node switches back to the application state, executing the new code.

(2) With delta encoding, the transferred delta size is usually small. However, the original Deluge protocol needs to disseminate data at the page granularity. Even when the data is far smaller than a page size, Deluge will transfer the entire page size (padding the data to the page size). We modify Deluge to transfer only the needed data.

(3) Although the Stream approach can address the problem of space limitations, it causes reliability issues. We encountered a problem in applying the original Stream approach in network reprogramming: a single broadcast of the state switching command is insufficient to enforce all nodes to switch to the same state, especially in networks with unreliable links.

To address this issue, we used a reliable dissemination service for disseminating the node state information. It works as follows:

—Nodes periodically exchange state information according to the Trickle algorithm.
—A node records state information including (a) its running state, (b) the state version number, and (c) the application version number. The running state refers to the reprogramming state or the application state. The state version number is used to synchronize the running states of all sensor nodes: a node should switch to the

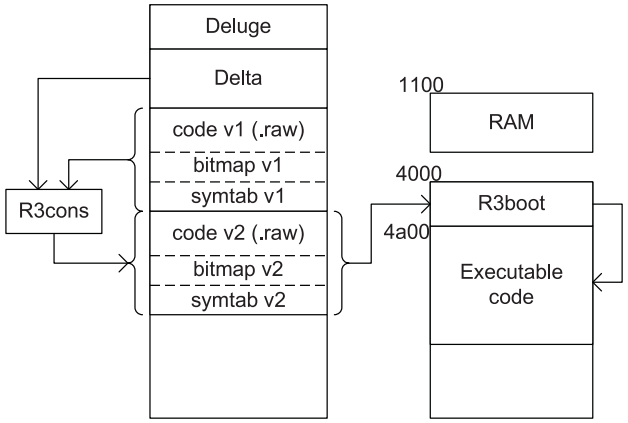


Fig. 6. Memory layout in R3.

running state with a newer state version number. The application version number is used to synchronize the application codes of all sensor nodes: when a new code version is found, a node should request to obtain the new code.

—Information of (a) and (b) is persistent across node reboots.

From an application programmer's perspective, we require that the application program code to be wired with a reprogramming support module that incorporates the above-mentioned optimizations (e.g., reliably disseminate state switching commands). If a node switches to the new application state while other nodes remain in the reprogramming state or the old application state, this node will advertise its current state (with its newer state version), causing other nodes to enter the new application state when possible. If there are nodes without the new application code, it will request to nodes that advertise having the new code. So, eventually, all network nodes will converge to execute the new application code. We would also emphasize that a node will switch to the new application code only when the base station issues the corresponding command (e.g., when reprogramming is finished). During the reprogramming state, nodes will receive and forward the new application code without executing the code.

In the worst case, a node can no longer respond to any wireless commands (due to software bugs). We can use the watchdog mechanism for recovery. In this case, the application code should additionally wire the watchdog component so that the application will reboot to the reprogramming image periodically (e.g., every three days). We assume that the reprogramming image is well tested and thus reliable. In this way, we ensure that sensor nodes can always respond to wireless commands in the reprogramming state. We can reprogram the network when the sensor nodes are in the reprogramming state or disseminate a command to enforce all nodes to switch to the original application state to continue the execution.

### 5.4. Reconstruction and Loading: R3cons and R3boot

As shown in Figure 6, R3 uses at least four volumes in the external flash. The first volume is used to store the reprogramming image. The second volume is used to store the received delta. The third and fourth volumes are used to store the program codes of two successive versions.

After receiving the delta file, the R3cons component (wired into the reprogramming image) is used to reconstruct the new code by patching the delta to the old code.

When a reboot command is received, our bootloader, R3boot, loads the reconstructed new code for execution. Because R3boot compiles to about 1.3KB, we allocate for it the address space 0x4000–0x4a00 (the same as tosboot). The application code space starts from 0x4a00.

It is worth noting that we directly burn the executable code onto the program flash when a node is first programmed. In addition, we place the program image (denoted as version 0) including "raw," bitmap, and symbol tables to the external flash such that the successive new program image (version 1) can be built on the external flash once a delta is received. R3boot loads the program image on the external flash to the program flash. The first programming stage differs from other reprogramming stages in that we directly generate executable code at the PC side (instead of using R3cons and R3diff at the sensor nodes) and burn the executable code onto the program flash at the sensor nodes.

## 6. COMPARISON AND ANALYSIS

In this section, we illustratively compare R3 with existing similarity preserving methods. We also build an analytical model to show the quantitative benefits of R3.

### 6.1. Illustrative Comparison of Different Similarity Preserving Methods

Figure 7 shows four similarity preserving methods for the change case in which the size of func1() increases to the address before 0x5400. We show two reference instructions with func2() as the target address. Before the change, func2() locates at 5000. After the change, func2() locates at 0x5400.

(1) The slop region approach [Koshy and Pandey 2005a] leaves a slop region of a few bytes after each function. When func1() increases within the slop region (e.g., before address 0x5000 as shown in the left part of Figure 7(a)), func2() does not shift (i.e., the start address 0x5000 of func2() will not change). Hence, calls to func2() do not change. However, when func1() increases beyond the slop region (e.g., to the address just before 0x5400 as shown in the right part of Figure 7(a)), func2() needs to be shifted (the start address of func2() now becomes 0x5400), causing calls to func2() to change correspondingly. Another major drawback is that it causes memory segmentations and thus inefficient use of memory.

(2) The function indirection approach employed by Zephyr and Hermes [Panta et al. 2009; Panta and Bagchi 2009] directs each function call to an indirection table entry (e.g., the entry locates at 0xf000) that calls the actual target function. Because the indirection table is placed at a fixed address in the program flash, function calls in the old and new code remain the same when the functions change. However, the problem is that this approach cannot preserve similarity in other reference instructions (e.g., br, mov) other than call. Figure 8 shows that, for the 10+ benchmarks we examined, the number of call instructions only occupies less than 50% of the total reference instructions. Also, it does not handle the shifts of data variables and interrupt service routines.

(3) R2 fills addresses in reference instructions to zeros and reserves two items (i.e., offset and addr) for each reference instruction. Although R2 achieves large similarity in the program code, it has a relatively large overhead in the relocation entries. For example, in Figure 7(c), the target addresses in the reference instructions are filled with zeros. For each reference instruction, there are two items in the relocation table, with the first one (offset) pointing to a memory address that needs relocation (e.g., memory addresses 0x4c00 and 0x4c80) and the second one (addr) pointing to the target address (e.g., function addresses 0x5000 or 0x5400). When the code is being loaded onto the program flash, a loader will use information in the relocation table to translate the target addresses in the reference instructions to the correct ones.

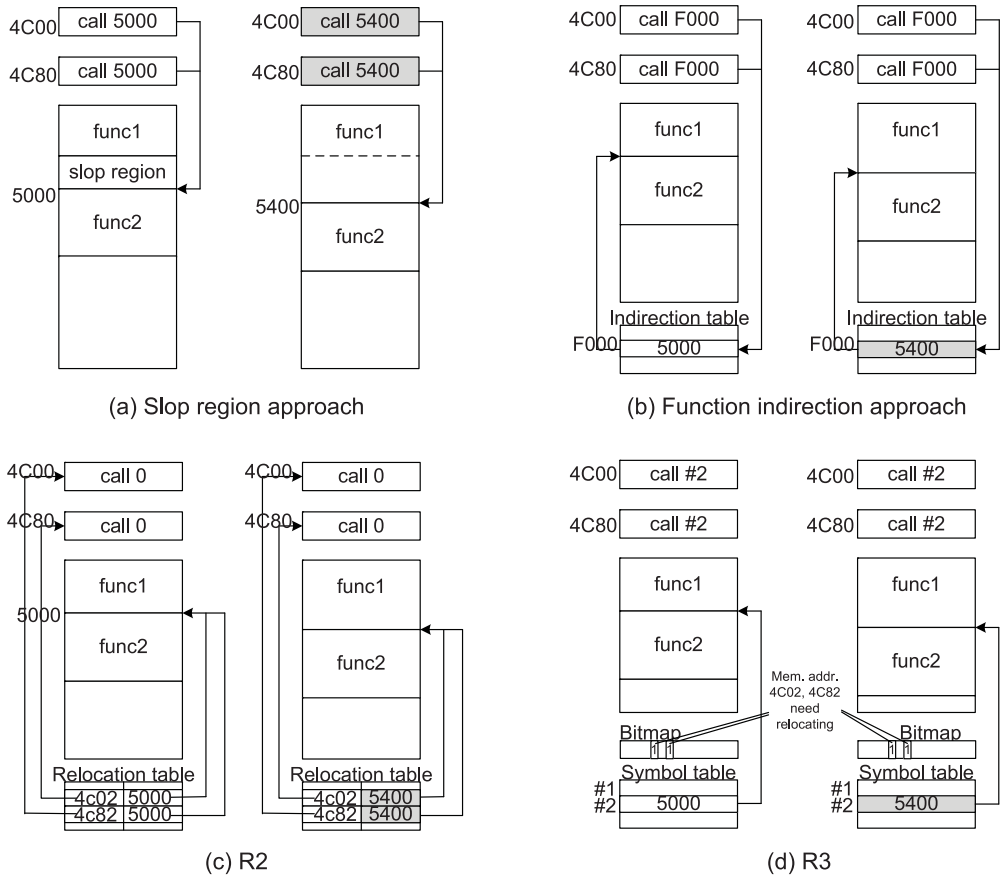


Fig. 7. Key ideas of three similarity preserving methods: (a) slop region, (b) function indirection, (c) R2, and (d) R3. (a) and (b) show the actual program memory layout. (c) and (d) show program images stored on the external flash. The program images are parsed and modified by R2boot or R3boot before they are loaded onto the program flash; for example, the "call #2" instruction in (d) is actually modified according to the bitmap and symbol table.

(4) Unlike R2, R3 fills addresses in reference instructions to the symbol indices (instead of zeros). For example, in Figure 7(d), the target addresses in the reference instructions are filled with 2 which refers to the symbol with index 2 in the symbol table. In the symbol table, each symbol is associated with its real target address. Another difference is that R3 uses a bitmap to indicate those memory locations that require relocation, eliminating the offset field in R2's relocation entry.

### 6.2. Analysis

We consider the example of code change shown in Figure 2. The following notations are used for our analysis.

—C denotes the code size in bytes.
—A denotes the total number of instructions.
—n denotes the number of reference instructions.
—m denotes the number of unique symbols.
—k denotes the number of call instructions.

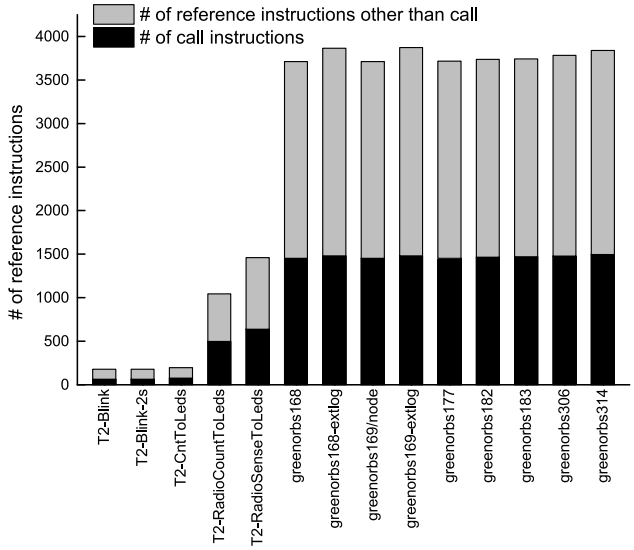


Fig. 8. Number of call instructions and all reference instructions.

The native incremental approach requires (n + 1) COPY commands and n ADD commands. Assume the address field in the reference instructions consumes 2 bytes; the delta size is thus $(n+1)\times\beta + n\times(\alpha+2) = 10n+5$.

Zephyr needs 10(n-k)+5 bytes because only the (n-k) non-call reference instructions need to be added. The metadata overhead consumes at most 2m bytes (since the metadata is also transferred by a delta, so the actual transferred overhead may be smaller than 2m). The delta size of Zephyr is 10(n-k)+5+M[2m], where M[x] denotes the delta size for the x-byte metadata.

R2 needs one COPY command, which consumes $\beta = 5$ bytes. Additionally, we need metadata for relocation. R2 needs at most 4n bytes for relocation entries (since each entry consumes 4 bytes). The delta size of R2 is thus 5+M[4n].

R3 needs one COPY command, which consumes $\beta = 5$ bytes. In addition, R3 needs at most 2m byte symbol entries and an additional bitmap of at most C/16 bytes The delta size of R3 is thus 5+M[2m+C/16].

We use tools in the GCC utilities (e.g., msp430-objdump, readelf, etc.) as well as an ELF parser written by ourselves to investigate all the benchmarks shown in Figure 8, obtaining the following statistics about the benchmarks (e.g., the total number of instructions, the number of unique symbols, etc.).

—What's the relationship between code size (C) and the total number of instructions (A)? We experimentally find that $A \approx 0.36C$.
—Among all instructions (A), how many instructions are reference instructions (n)? We experimentally find that $n \approx 0.24A \approx 0.0864C$.
—Among all reference instructions (n), how many call instructions (k) are there? We experimentally find that $k \approx 0.4n \approx 0.034C$.
—Among all reference instructions (n), how many unique symbols (m) are there? We experimentally find that $m \approx 0.27n \approx 0.0233C$.

Without loss of generality, we consider M[x] = x/2. Table I shows the delta size in terms of key code features, as well as the approximated values with respect to the code size.

Table I. Delta Sizes for the Example Shown in Figure 2
M[x] denotes the delta size for the x-byte metadata; B[x] denotes the delta size for the x-byte bitmap in R3; $0 \leq M[x], B[x] \leq x$.

| Approach | Delta size | Approx value |
|---|---|---|
| Native inc | 10n+5 | 0.864C+5 |
| Zephyr | 10(n-k)+5+M[2m] | 0.54C+5 |
| R2 | 5+M[4n] | 5+0.17C |
| R3 | 5+M[2m+C/16] | 5+0.054C |

We can see that R3 achieves the smallest delta size.

## 7. EVALUATION

In this section, we evaluate the R3 reprogramming system. Section 7.1 describes the evaluation methodology. Sections 7.2 through 7.5 evaluate R3 in four aspects. Section 7.6 presents the reprogramming energy analysis. Section 7.7 presents an analysis on the network lifetime.

### 7.1. Methodology

We evaluate R3 on TinyOS/TelosB. We select five software change cases from the TinyOS distribution, as well as three software change cases in the development of GreenOrbs—a real sensor network system.

—Case 1: We change the Blink application from blinking the red LED every 1 second to blinking the red LED every 2 seconds.
—Case 2: We change the Blink application to CntToLeds, which increments a counter every 1 second and displays the lowest three bits on the LEDs.
—Case 3: We change the RadioCountToLeds application to RadioSenseToLeds.
—Case 4: We consider the GreenOrbs application of SVN version 168. The base version performs sensing, transmissions, and the like, while the updated version disables sensing and enables local logging on the external flash.
—Case 5: We consider the change case of GreenOrbs from version 182 to 183. The updated version fixes a link estimation bug in the 4bitle component.
—Case 6: We consider the change case of GreenOrbs from version 306 to 314. The updated version provides another parameter for reconfiguration.
—Case 7: We consider the change case of the Drip dissemination protocol (TestDrip) in TinyOS 1.x to that in TinyOS 2.0.2.
—Case 8: We consider the change case of the data collection protocol (TestNetwork) in TinyOS 2.0.2 to that in TinyOS 2.1.0.

It is worth noticing that Cases 4–6 are real software change cases from our project GreenOrbs [Mo et al. 2009]. The GreenOrbs application compiles to nearly 45KB and contains full functionalities, including data sensing, collection, parameter dissemination, and the like. Cases 7 and 8 consider major changes in the TinyOS core (i.e., from version 1.x to 2.0.2, and from 2.0.2 to 2.1.0).

### 7.2. Differencing Algorithm

We first compare the delta sizes of three algorithms (i.e., Rsync, RMTD, and R3diff). All algorithms are run on a PC with CPU Quad 2.66GHz, 4GB RAM. Rsync is an open source project implemented in java. Both RMTD and R3diff are implemented in C and are compiled with GCC 4.1.

We generate delta without similarity preserving methods. Rsync is a block-based differencing algorithm that requires specifying the block size as a parameter. The original Rsync algorithm generates commands for every block. If the block size decreases,

Table II. Delta Sizes (in Bytes) of the diff Algorithms
In the notation "Rsync-*x*", *x* denotes the block size in bytes.

| Case | Rsync-32 | Rsync-16 | Rsync-8 | Rsync-4 | RMTD | R3 |
|---|---|---|---|---|---|---|
| 1 | 45 | 29 | 151 | 782 | 18 | 15 |
| 2 | 1,680 | 1,328 | 1,232 | 1,775 | 930 | 929 |
| 3 | 15,749 | 14,773 | 14,447 | 18,195 | 12,365 | 12,345 |
| 4 | 39,732 | 36,191 | 35,338 | 48,265 | 28,506 | 28,425 |
| 5 | 17,447 | 13,225 | 14,434 | 32,956 | 6,787 | 6,782 |
| 6 | 31,968 | 26,994 | 26,285 | 40,266 | 16,864 | 16,838 |
| 7 | 14,211 | 13,832 | 13,797 | 16,350 | 13,177 | 13,116 |
| 8 | 29,489 | 27,166 | 26,621 | 35,445 | 21,795 | 21,717 |

Table III. Execution Times (Seconds) of RMTD and R3diff on a PC

| Case | RMTD-core | RMTD-CS | R3diff |
|---|---|---|---|
| 1 | 0.01 | 0.20 | 0.01 |
| 2 | 0.01 | 0.19 | 0.01 |
| 3 | 0.33 | 6.04 | 0.04 |
| 4 | 5.61 | 55.47 | 0.64 |
| 5 | 5.27 | 54.46 | 0.32 |
| 6 | 5.86 | 55.66 | 0.29 |
| 7 | 0.11 | 4.47 | 0.02 |
| 8 | 2.38 | 51.69 | 0.15 |

the number of commands will increase. We optimize Rsync by combining commands with continuous bytes. For RMTD, we specify $\beta' = 5$ and calculate the delta size by examining the command sequence under $\alpha = 3$ and $\beta = 5$. We can see from Table II that both R3diff and RMTD generate much smaller delta sizes than Rsync. Although R3diff and RMTD generate deltas with roughly the same size, the real figures of R3diff are consistently smaller than RMTD.

We then compare the execution times of R3diff and RMTD in Table III. RMTD's execution time consists of two major components. The core component executes a dynamic programming algorithm while another component, we call it RMTD-CS, is used to search and store the common segments between two code versions. We find that the search and store process can be quite long for large program code. For all the benchmarks we examined, R3diff's execution times are less than 1 second. For small programs and small workloads, the execution time of the differencing algorithm is not a very important factor. However, it will be important when the delta will be frequently generated in a central server (e.g., a software server where a large number of apps with different versions are hosted). It is also important in applying differencing algorithms in other fields (e.g., versioning control where a SVN server hosts a large number of projects with numerous versions).

Compared with the execution time, the reduction in memory consumption is more important. We compare the memory consumption of R3diff and RMTD in Table IV. RMTD's memory consumption consists of two components. The core component uses memory for storing the common segments, the *opt* array, and additional arrays for backtracking. The RMTD-Table component uses memory for storing a two-dimensional array for searching the common segments. We find that the memory consumption of RMTD-Table is significantly larger than RMTD-core. R3diff's memory consumption is slightly larger than RMTD-core because we need additional arrays for $opt^A$ and $opt^C$ and for backtracking purpose. In terms of total memory consumption, R3diff is much smaller than RMTD.

Table IV. Memory Consumption (Bytes) of RMTD and R3diff on a PC

| Case | RMTD-core | RMTD-Table | R3diff |
|---|---|---|---|
| 1 | 19,198 | 916,318 | 54,084 |
| 2 | 19,402 | 913,614 | 53,988 |
| 3 | 168,794 | 26,149,500 | 305,116 |
| 4 | 749,626 | 237,517,190 | 877,092 |
| 5 | 742,434 | 227,116,532 | 852,556 |
| 6 | 793,294 | 236,761,400 | 872,196 |
| 7 | 101,666 | 22,043,136 | 270,436 |
| 8 | 489,637 | 123,195,726 | 632,236 |

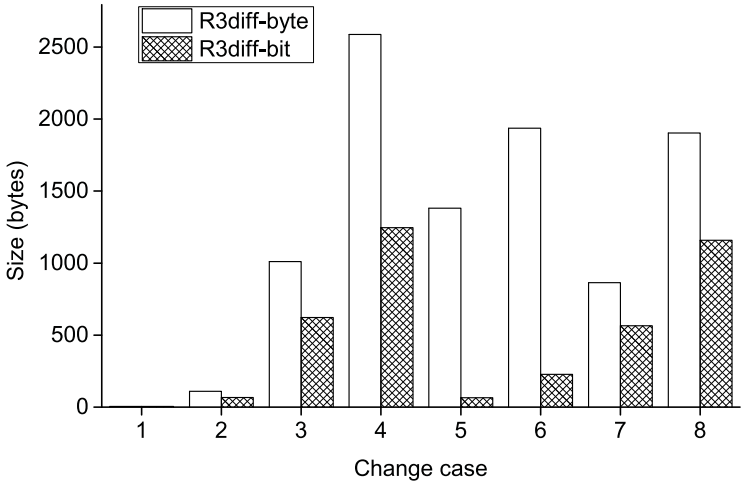


Fig. 9. Bit-level comparison vs byte-level comparison.

Finally, we show that bit-level comparison is effective in reducing the delta size of the bitmap files. As Figure 9 shows, bit-level comparison on the bitmap files within R3diff results in 40% to 90% reductions. It is worth noting that there are no bars for change case 1 because the corresponding values are very small: both R3diff-byte and R3diff-bit generate deltas of 5 bytes.

### 7.3. Delta Size

This section evaluates how similarity preserving methods affect the delta size. When possible, we use these methods with R3diff, which is proved to be optimal. It is worth noting that different methods may require slightly different differencing algorithms for improved performance. For example, Hermes requires additional metacommands to optimize the delta of the function indirection table, R2 requires a content-aware differencing algorithm to optimize the delta of the relocation table, and R3 requires a bit-level comparison algorithm to optimize the delta of bitmap files. In the evaluation, we use the improved differencing algorithms for Hermes, R2, and R3.

Figure 10 shows the transferred file sizes for various reprogramming approaches, including Stream, Zephyr, R2/R2c, and R3. Note that R2c is a variation of R2 with the chained reference technique described in Section 5.1. In Figure 10, we also show the sizes of metadata. The notation "Table" refers to the indirection table in Zephyr, the relocation table in R2, or the symbol table in R3. The notation "bitmap" refers to the bitmap file in R3. We can see that R3 greatly reduces R2's metadata overhead while preserving as high a code similarity as R2, resulting in the smallest delta size.

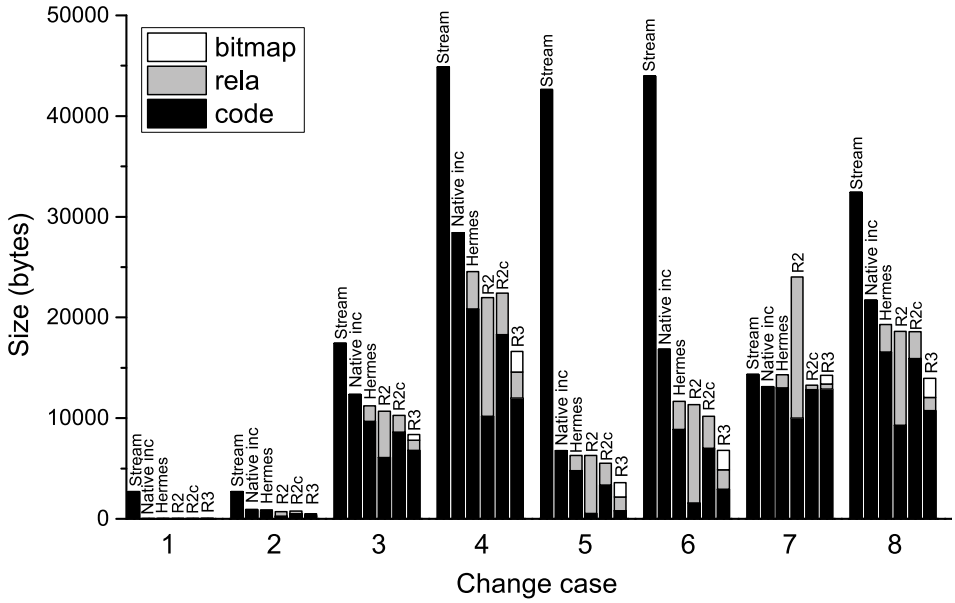


Fig. 10. Delta size comparison among six approaches.

We notice that a particular incremental approach may be effective in some cases while be less effective in other cases. It would be interesting to see under what circumstances a particular approach will be effective. We thus propose a simple metric called similarity index (between two program versions) to study to what extent a particular approach preserves the similarity. Let $R_1$ be the number of reference instructions in the old code and $R_2$ be the number of reference instructions in the new code. $R_m$ is the number of matching reference instructions between the two codes. Two reference instructions are called matched if they make references to the same symbol, and the symbol addresses in the reference instructions are the same in two versions. We define: $SI = \frac{R_m}{\max(R_1, R_2)}$. In the native incremental approach, the symbol address in the reference instruction is the actual target address. Because the symbol's target address changes because of code shift, SI for the native incremental approach is expected to be low. On the other hand, in R2, all symbol addresses are filled with 0, whereas in R3 all symbol addresses are replaced by the symbol indices. Hence, the SIs will be high. Figure 11 shows the SIs of different approaches for the eight benchmarks. We can see two facts: (1) The SI metric can roughly reflect the effectiveness of each approach. For example, in cases 3 and 4, the SIs for the native incremental approach are low. Hence, the native incremental approach is expected to have low performance (this is confirmed in Figure 10). (2) Both R2 and R3 achieve the largest similarity.

It is also interesting to note that although the SIs of R2 and R3 are the same, R2 achieves a smaller code delta. The reason is explained as follows: (1) R3 fills the symbol address to the symbol index so that reference instructions to the same symbol are guaranteed to be the same. (2) R2 fills all symbol addresses to 0 so that reference instructions to different symbols might be the same. Hence, R2 achieves a smaller code delta size at the cost of larger metadata overhead. Consider the code change case shown in Figure 13, where code blocks 1, 2, and 3 contain the same set of instructions. With R2, the entire code segment can be copied because the target addresses in the call instructions are all filled with zeros. With R3, however, the target addresses in

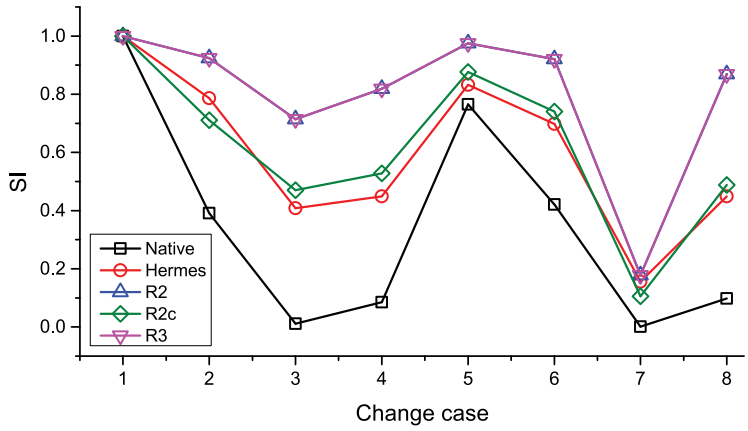


Fig. 11. The SI metric.

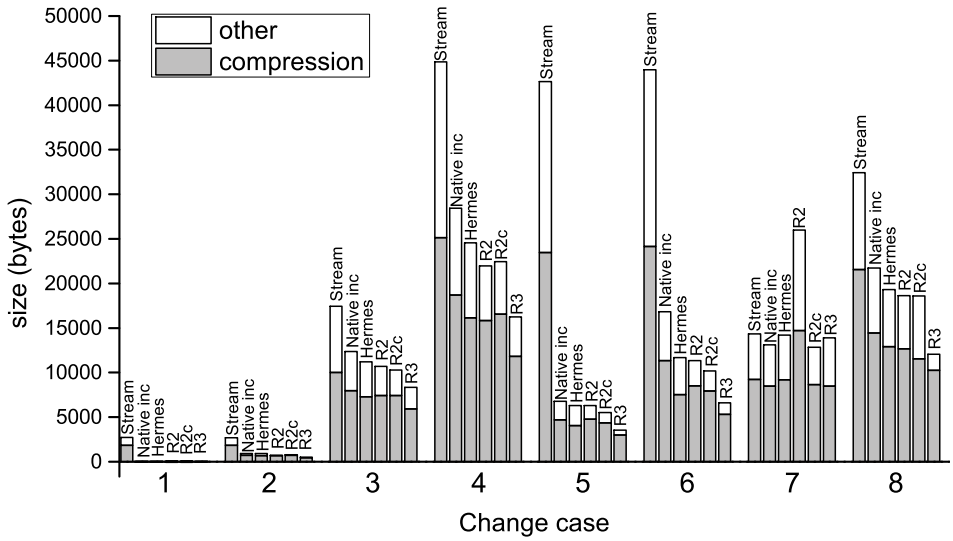


Fig. 12. Incremental approaches with compression.

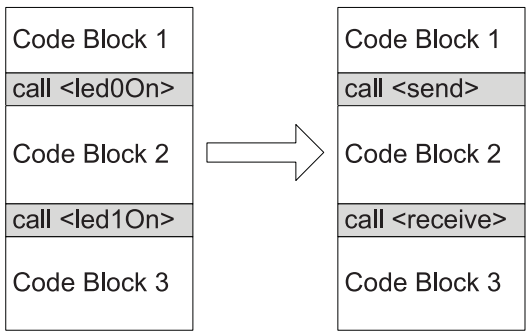


Fig. 13. A code change example.

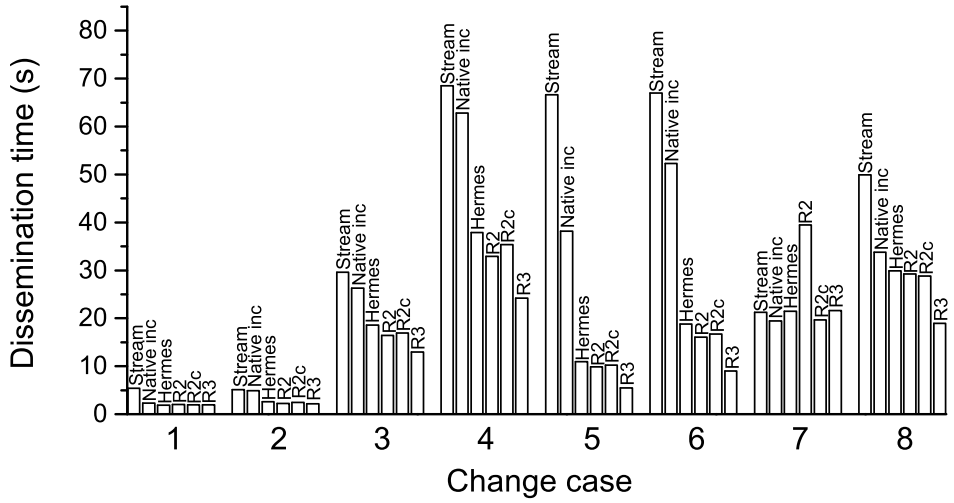


Fig. 14. Dissemination time in the single experiment.

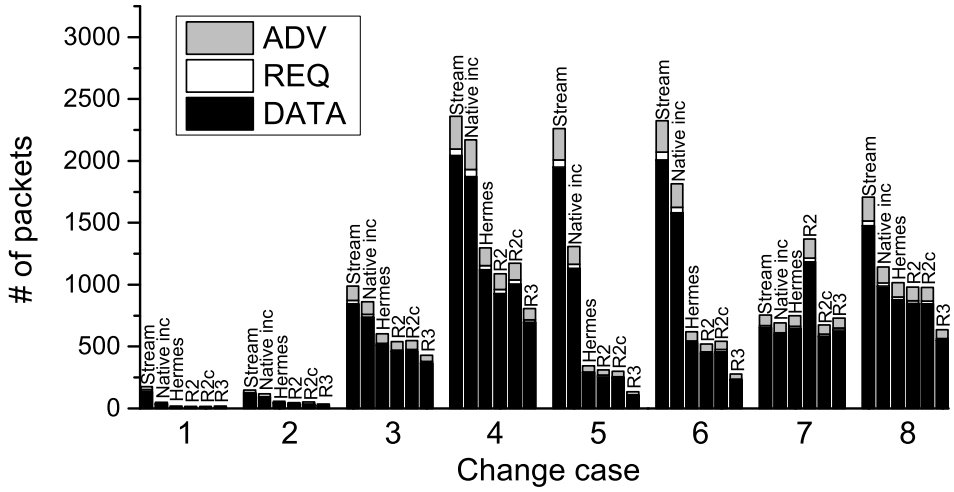


Fig. 15. Dissemination traffic in the single experiment.

the call instructions are different because they refer to different symbols (i.e., different semantics). From this example, we can see that R2 can achieve a smaller delta size.

We also investigate the effects of an incremental approach combined with compression. We can see from Figure 12 that an incremental approach combined with gzip compression can further reduce the transferred file size. Although large files are more compressible, R3 combined with compression generates the smallest file size.

### 7.4. Dissemination Performance

In this section, we examine how the delta size impacts the dissemination performance.

Figure 14 shows the dissemination times, and Figure 15 shows the number of transmitted packets in a single-hop network with 5 TelosB nodes. Figure 16 shows the dissemination times, and Figure 17 shows the number of transmitted packets in a multihop network with 35 TelosB nodes.

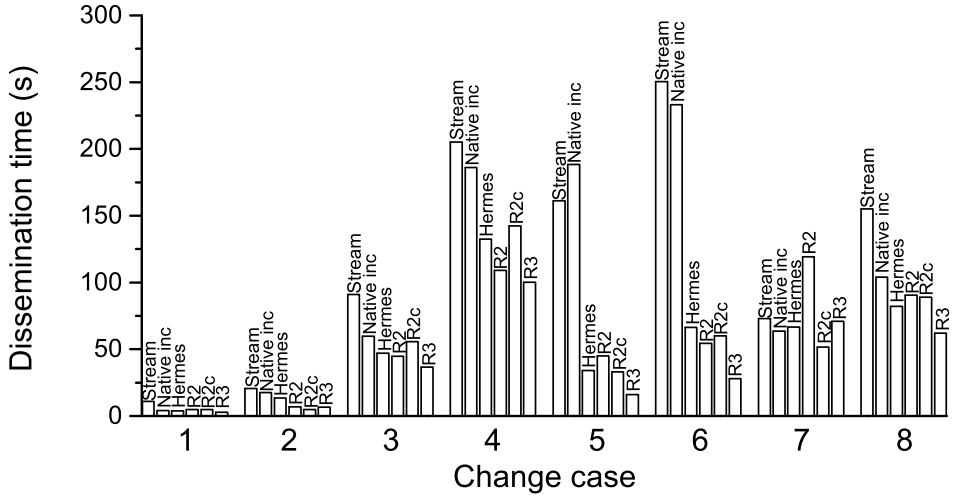


Fig. 16. Dissemination time in the multihop experiment.

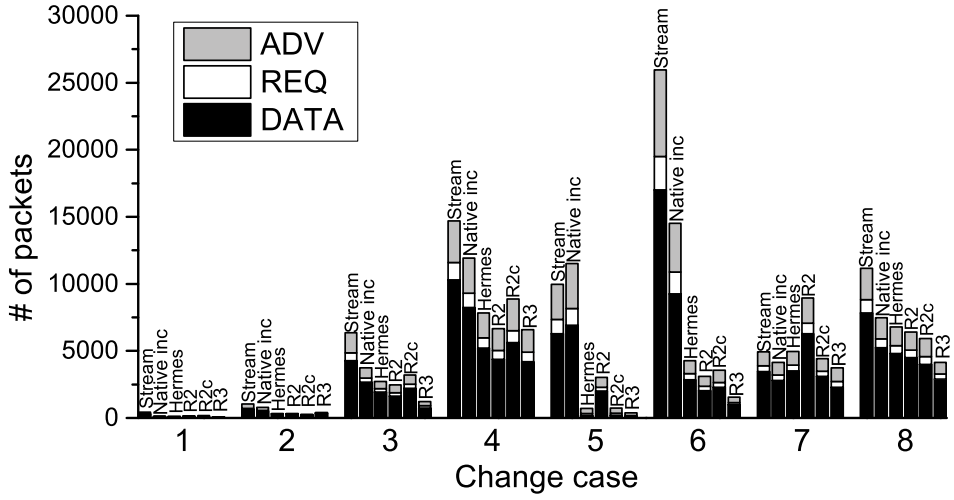


Fig. 17. Dissemination traffic in the multihop experiment.

We can see that the smaller the delta size, the more efficient the dissemination process. If we denote $t_p^s$ and $t_p^m$ as the average time to receive one byte during dissemination in the single hop experiment and the multihop experiment, respectively, we get $t_p^s \approx 1.5$ms/byte and $t_p^m \approx 4.68$ms/byte by linear fitting.

### 7.5. Reconstruction and Loading Overhead

In this section, we evaluate the reconstruction and loading overhead (at the sensor node side) in terms of memory consumption and execution time.

The R3cons component is wired into the reprogramming image. The reprogramming image mainly consists of the Deluge protocol. The RAM and ROM overhead of the entire reprogramming image is 1,378 bytes and 36,154 bytes, respectively. The RAM and ROM overheads of the R3cons component are 468 bytes and 7,586 bytes, respectively. The R3cons component uses a 100-byte read cache. We also devise an optimized R3cons

Table V. Execution Times (s) of R3cons and R3boot on TelosB

| Case | R3cons | R3cons-opt | R3boot | tosboot |
|---|---|---|---|---|
| 1 | 0.405 | 0.395 | 0.200 | 0.148 |
| 2 | 0.480 | 0.406 | 0.194 | 0.148 |
| 3 | 4.140 | 2.800 | 1.080 | 0.744 |
| 4 | 12.000 | 7.500 | 2.590 | 1.720 |
| 5 | 7.470 | 6.190 | 2.620 | 1.750 |
| 6 | 8.600 | 6.540 | 2.660 | 1.800 |
| 7 | 4.960 | 3.580 | 0.890 | 0.608 |
| 8 | 8.230 | 5.830 | 1.950 | 1.313 |

component (R3cons-opt) that additionally uses a 100-byte write cache. The RAM and ROM overheads of R3cons-opt are 568 bytes and 9,912 bytes, respectively. It is worth noting that the overhead of R3cons does not affect applications because it is wired into the reprogramming image.

The RAM and ROM overheads of R3boot are 0 and 1,270 bytes, respectively. R3boot only uses local variables. For comparison, the RAM and ROM overheads of tosboot are 0 and 1,776 bytes, respectively. Similar to tosboot, we allocate the address space 0x4000–0x4a00 to R3boot.

The execution times of R3cons and R3boot are shown in Table V. We can see that the operations in R3cons and R3boot are quite time-consuming compared to other operations in sensor nodes (which are usually on the order of ms). This is because I/Os in the external flash and program flash are costly. These results illustrate that there may exist a tradeoff between delta compressibility and the processing complexity on the nodes. We discuss this tradeoff in the next subsection.

### 7.6. Reprogramming Energy Analysis

We develop a model to study the overall energy consumption during reprogramming. The energy consumption during reprogramming, $E_{\text{reprog}}$, can be computed as follows.

$$E_{\text{reprog}} = E_{\text{sw}} + E_{\text{p}} + E_{\text{s}} + E_{\text{dc}} + E_{\text{rc}} + E_{\text{ld}}, \tag{3}$$

where $E_{\text{sw}}$ is the overhead to switch to the reprogramming image, $E_{\text{p}}$ is the energy consumption during dissemination, $E_{\text{s}}$ is the energy consumption to store the transferred file on the external flash, $E_{\text{dc}}$ is the energy consumption to decompress the transferred file if it is compressed, $E_{\text{rc}}$ is energy consumption to reconstruct the new code if incremental reprogramming is used, and $E_{\text{ld}}$ is the energy consumption to load the code to the program flash. It is worth noting that we use the image switching mechanism similar to Stream [Panta et al. 2007] to reduce the dissemination cost. The key idea is to separate the application image and reprogramming image so that the reprogramming image does not need to be wired into the application image. When reprogramming is required, a node switches to the reprogramming image for receiving the new code. After receiving the new code, the node switches back to the application image, executing the new code.

We summarize the parameters we used in Table VI. We explain some parameters as follows: $t_{\text{p}}$ denotes the average time to receive 1 byte during dissemination. For our single-hop experiment (see Section 7.4), $t_{\text{p}} = t_{\text{p}}^{s} = 1.5$ms/byte. Because $I_{\text{rx}} \approx I_{\text{tx}}$, we set $I_{\text{radio}}$ to 20mA to represent the average current when the radio is on in order to simplify our analysis. $t_d$ denotes the time required for decompressing 1 byte. $t_d \approx 1.26$ms/byte in R3's implementation. This number is consistent with the implementation of the gzip algorithm in Tsiftes et al. [2008].

Table VI. Parameter Settings (for TelosB Nodes)

| Notation | Value | Meaning |
|---|---|---|
| $U$ | 3V | voltage |
| $I_{\text{read}}$ | 5mA | current when reading bytes from external flash |
| $I_{\text{wrt}}$ | 12mA | current when writing bytes to external flash |
| $I_{\text{prog}}$ | 3mA | current when writing bytes to the program flash |
| $I_{\text{load}}$ | 4mA | average current when loading code to program flash |
| $I_{\text{tx}}$ | 19.5mA | current when the radio is in transmission mode (0dBm) |
| $I_{\text{rx}}$ | 21.8mA | current when the radio is in reception mode |
| $I_{\text{mcu}}$ | 1.8mA | current when the MCU is active |
| $t_{\text{read}}$ | 0.056ms | average time to read one byte from external flash |
| $t_{\text{wrt}}$ | 0.059ms | average time to write one byte to external flash |
| $t_{\text{load}}$ | 0.045ms | average time to load one byte from external flash to the program flash |
| $s_{\text{p}}$ | depends on app | size of the transferred file (i.e., delta for incremental reprogramming). |
| $s_{\text{app}}$ | depends on app | size of the new application code. |
| $s_{\text{rela}}$ | depends on app | size of the relocation information. |
| $t_{\text{p}}$ | | average time to receive one byte during dissemination, e.g., $t_{\text{p}} \approx 1.5$ms/byte in our single hop experiment (Section 7.4). |
| $s_{\text{reprog}}$ | | size of the reprogram image. According to the Deluge implementation in TinyOS 2.x, $s_{\text{reprog}} \approx 37$KB. |
| $t_{\text{d}}$ | | time required for decompressing one byte. $t_{\text{d}} \approx 1.26$ms/byte. |

We have the following equations:

$$E_{\text{sw}} = U I_{\text{load}} \cdot t_{\text{load}} s_{\text{reprog}} \tag{4}$$

$$E_{\text{p}} = U I_{\text{radio}} \cdot t_{\text{p}} s_{\text{p}} \tag{5}$$

$$E_{\text{s}} = U I_{\text{wrt}} \cdot t_{\text{wrt}} s_{\text{p}} \tag{6}$$

$$E_{\text{dc}} = U(I_{\text{read}} \cdot t_{\text{read}} s_{\text{p}} + I_{\text{wrt}} \cdot t_{\text{wrt}} s_{\text{app}} + I_{\text{cpu}} t_{\text{d}} s_{\text{p}}) \tag{7}$$

$$E_{\text{rc}} = U(I_{\text{read}} \cdot t_{\text{read}} s_{\text{app}} + I_{\text{wrt}} \cdot t_{\text{wrt}} s_{\text{app}}) \tag{8}$$

$$E_{\text{ld}} = U(I_{\text{load}} \cdot t_{\text{load}} s_{\text{app}} + I_{\text{read}} \cdot t_{\text{read}} s_{\text{rela}}) \tag{9}$$

With this energy model, we are interested in answering the following two questions:

—How does the delta size affect the overall reprogramming energy consumption? Figure 18 shows the reprogramming energy consumption of different approaches for the eight change cases. Overall, the number of reduced bytes results in a similar reduction in the overall reprogramming energy consumption. This is because the energy consumption during dissemination consumes the largest portion. In Figure 19, we investigate the reprogramming energy when $t_{\text{p}} = 0.1$ms/byte (i.e., almost approaching the rate limit of the 250kbps CC2420 radio under CSMA MAC). We can see several facts from Figure 19. First, the image switching overhead poses a baseline energy consumption. Second, dissemination still consumes a large portion of energy, especially for large software change cases. Third, the flash I/O is also energy hungry. By reducing the transferred file size, we also reduce the energy during flash I/O.

—Can compression further reduce the reprogramming energy consumption, and to what extent if it can? Figures 20 and 21 compare the energy consumption of different approaches and the corresponding approaches with compression. In these figures, the reduced energy $= E_{\text{reprog}}$[an approach] $- E_{\text{reprog}}$[the corresponding approach with

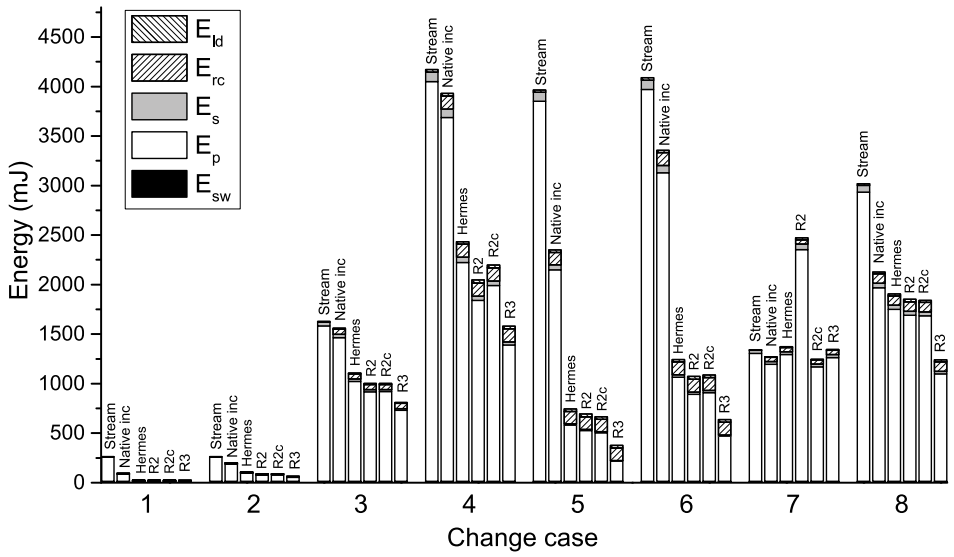


Fig. 18. Reprogramming energy consumption of different components ($t_p = 1.5$ms/byte).

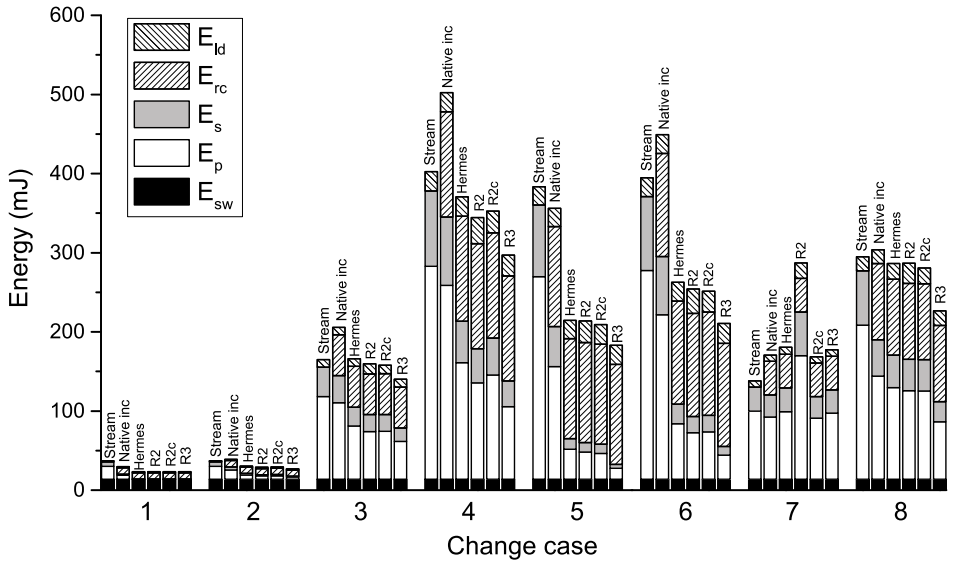


Fig. 19. Reprogramming energy consumption of different components ($t_p = 0.1$ms/byte).

compression]. Thus, a positive value indicates compression can reduce the energy consumption and vice versa. Figures 20 and 21 show the results with $t_{\mathrm{p}} = 1.5$ms/byte and $t_{\mathrm{p}} = 0.1$ms/byte, respectively. We can see that (1) compression is effective when the transferred file is large and compressible, but ineffective when the transferred file is small and incompressible. (2) Compression will be ineffective when the radio transmission rate is fast enough (e.g., approaching to the 250kbps limit under CSMA MAC).

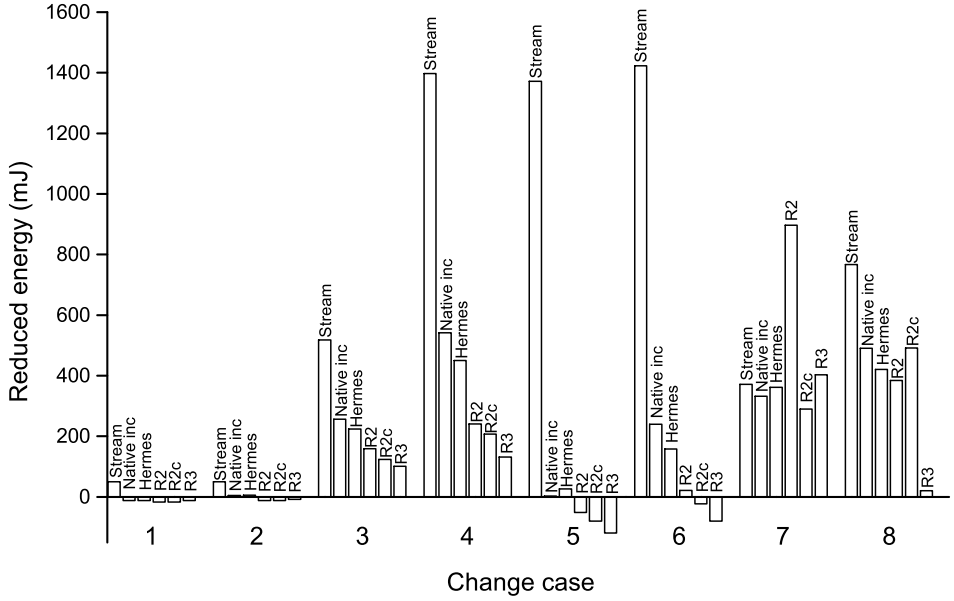


Fig. 20. Reduction of reprogramming energy compared to approaches combined with compression ($t_p$ = 1.5ms/byte).

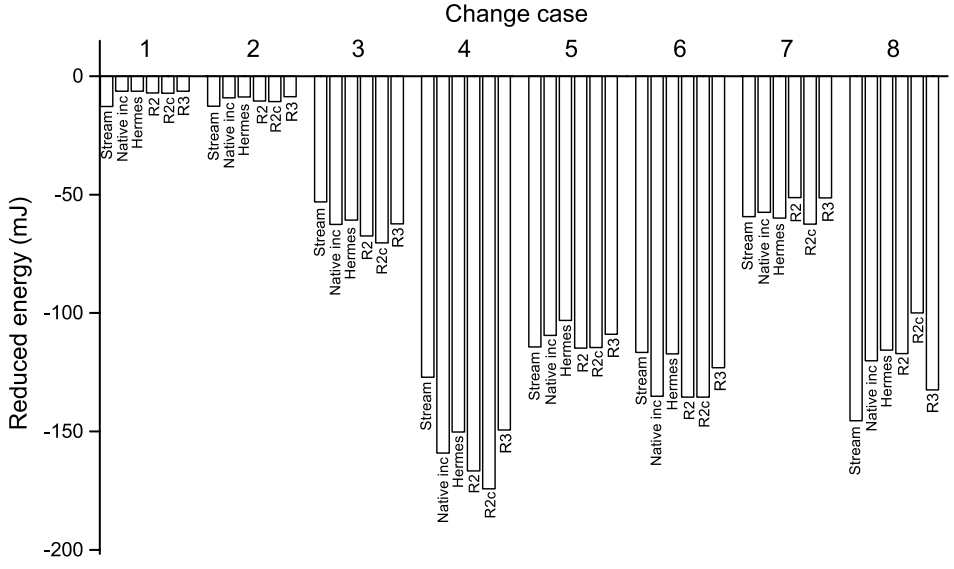


Fig. 21. Reduction of reprogramming energy compared to approaches combined with compression ($t_p$ = 0.1ms/byte).

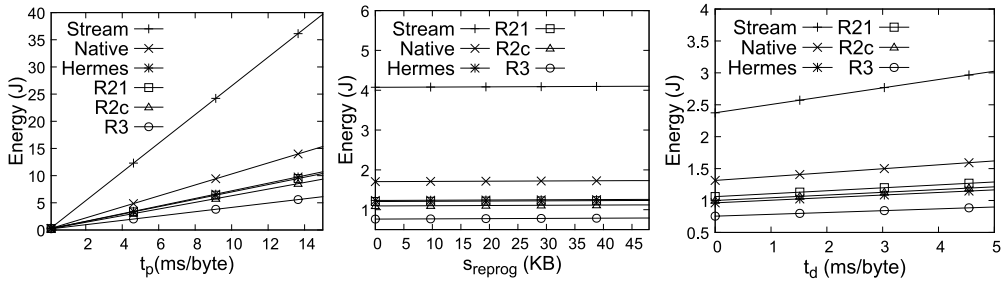


Fig. 22. Reprogramming energy with varying (a) $t_p$, (b) $s_{\text{reprog}}$, and (c) $t_d$.

It is also important to understand the reprogramming energy consumption with varying parameter settings. In the following, we investigate how the reprogramming energy consumption varies with different values of $t_p$, $s_{\text{reprog}}$, and $t_d$. Without loss of generality, we consider the software change in case 6.

Figure 22(a) shows the reprogramming energy with varying $t_p$. $t_p$ captures how fast the code/delta is disseminated in the network. Clearly, $t_p$ has a large impact on the reprogramming energy. For example, in a large-scale network with lossy links, it is expected that the code dissemination will be long (i.e., $t_p$ is large). Hence, the reprogramming energy consumption will also be large.

Figure 22(b) shows the reprogramming energy with varying $s_{\text{reprog}}$. We note that $s_{\text{reprog}}$ is the code size of the reprogramming image. Following Stream's optimization, the reprogramming image is not involved in our code dissemination process. It is only involved in the state switching process. Therefore, the impact of $s_{\text{reprog}}$ on the reprogramming energy is small.

Figure 22(c) shows the reprogramming energy with varying $t_d$, which is the time for decoding a byte. Clearly, slow decoding will incur relatively large energy consumption.

### 7.7. Network Lifetime Analysis

In this section, we continue to analyze the lifetime of different approaches.

We use several key system parameters to model the long-term energy efficiency of a sensor system with network reprogramming. The average working period between two reprogramming actions is denoted as $t$ (days). The sum of reprogramming energy consumption and the execution energy consumption during $t$ can be approximated as

$$E_t = Uf(I_{\text{radio}} + I_{\text{mcu}})t + E_{\text{reprog}}, \tag{10}$$

where $f$ is the duty cycle of the sensor network under normal operations, and $E_{\text{reprog}}$ is the energy consumption during reprogramming.

Batteries exhibit self-discharge phenomenon. The energy consumption due to self-discharge can be calculated as follows:

$$B'(L) = rLB, \tag{11}$$

where $r$ is the self-discharge rate (% per day), $L$ is the time in terms of days, and $B = 2,200$mAh is the battery capacity. Considering that standard NiMH batteries lose one third of their charge after one year, we assume $r = \frac{0.33}{365}$.

We can get the following equation:

$$UB = E_t \cdot \frac{L}{t + t_p \cdot s_p} + UB'(L). \tag{12}$$

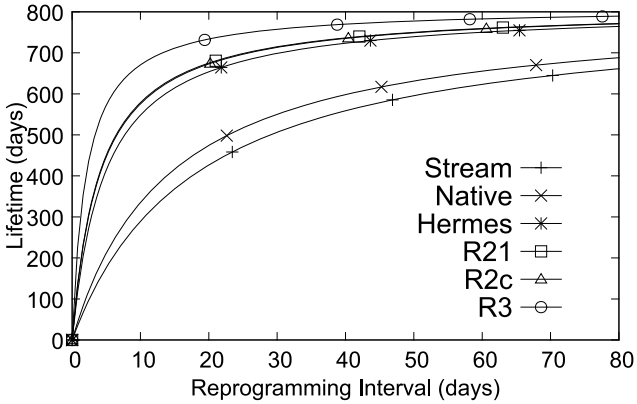


Fig. 23. Comparison of six reprogramming approaches in terms of network lifetime.

The lifetime of the network can thus be calculated as follows:

$$L(t) = \frac{UB}{\frac{E_t}{t+t_p \cdot s_p} + rUB.} \tag{13}$$

Note that $E_t$ roughly represents the energy consumption during $t$ plus the time for dissemination, $t_p \cdot s_p$.

Note that $t_p$ can be quite large in large-scale sensor networks with lossy links. According to Dong et al. [2013], $t_p \approx 200$s/KB in a $10 \times 10$ networks with 10-foot internode spacing. Because sensor networks typically work under extreme low duty cycles, the duty cycle $f$ is set to 0.1% according to Liang et al. [2008].

Figure 23 shows how the network lifetime varies with different reprogramming intervals. As expected, during the development of WSN when updates are frequent (i.e., $t$ is small), the benefit of R3 is larger. After maturing, when updates are infrequent (i.e., $t$ is large), the benefit of R3 is smaller. Although simple reprogramming approaches might be favored for mature systems, we think our approach will be important for outdoor sensor network testbeds [Shu et al. 2008; Prabal et al. 2006] where wireless reprogramming will be performed frequently [Dunkels et al. 2006]. It is also worth noticing that a long reprogramming time may also interrupt the normal operation of a sensor network. Therefore, we usually require the dissemination process to take place in as short a time as possible. There are a number of relevant works in the literature to shorten the dissemination time [Kulkarni and Wang 2005; Hagedorn et al. 2008; Dong et al. 2011, 2011; Gao et al. 2013], resulting in a reduction factor of 1.21 to 2.42 with respect to the default Deluge protocol. Our work shortens the completion time by reducing the code size, resulting in a reduction factor of 2 to 100 with respect to Stream and 1.25 to 1.67 with respect to R2.

## 8. DISCUSSION

This section discusses several important issues about R3.

### 8.1. Consideration for Other Platforms

We implement R3 on the MSP platform, it would be interesting to see how scalable it is by porting it to other platforms like AVR. AVR has 19 relocation types, whereas MSP has only one. For R3, each bit in the bitmap must be extended to more bits in order to encode the relocation types. On AVR platforms, relocation is performed at 4-byte granularity. If we reserve 5 bits (to encode all 19 relocation types and the additional

type indicating no relocation) for each memory location at the 4-byte boundary, the bitmap overhead on AVR platforms would be 2.5 times the overhead for MSP platforms. We can selectively preserve a subset of relocation types in order to preserve a high code similarity while at the same time maintaining a reasonable metadata overhead. For example, if we reserve 2 bits for each possible memory location and preserve the three most frequently occurred relocation types, the bitmap overhead would be the same as the MSP platform. If we reserve 3 bits for each possible memory location and preserve the seven most frequently occurred relocation types, the bitmap overhead would be 1.5 times overhead for MSP platform. After investigating typical benchmarks, we find that the number of reference instructions of the top three relocation types accounts for 64% to 80% of the number of all reference instructions, and the number of reference instructions of the top seven relocation types accounts for 80% to 90% of the number of all reference instructions. This implies that a relatively high code similarity can still be preserved with a reasonably low metadata overhead.

R3's design will not be affected by the particular feature that often requires program flash memory of MCUs to be written in pages. This is because R3 reconstructs the new code on the external flash before loading the code onto the program flash. In the loading process, the I/O drivers of the external flash, as well as the program flash, will employ local buffer mechanisms that meet the underlying requirements of the MCU.

### 8.2. Consideration for Compilation Optimization

The R3 similarity preserving method works at the binary level. We believe R3 works correctly under compiler optimization, but its performance might be affected due to different compiler optimization techniques.

For example, some compilers can compress program code size significantly through removal of redundant code. In general, this will also result in a smaller delta compared with the case without compilation optimization. The relative benefits of incremental reprogramming compared to full image replacement, however, would be smaller because small files are generally less compressible.

As another example, some compilers perform function inlining, which will increase the code size. The relative benefits of incremental reprogramming compared to full image replacement would be larger. We expect that there would be fewer reference instructions in the program due to fewer functions. Therefore, the relative benefits of all similarity preserving methods (e.g., R3, R2, and Hermes) compared to native incremental reprogramming would be smaller.

### 8.3. Consideration for Heterogeneity

In heterogeneous sensor networks with identical hardware and different program code, the base station must have knowledge of the exact software configuration of the sensor nodes on which the delta script is to be run [Dunkels et al. 2006]. For example, in a network with different program codes for several types of sensor nodes (e.g., normal nodes, cluster heads, etc.), the base station can generate different delta scripts and employ a multicast dissemination protocol to disseminate delta scripts to the corresponding nodes. In Dunkels et al. [2006], the authors also mentioned that if the software heterogeneity is caused by different compiler versions or different OS versions, incremental reprogramming approaches do not scale since the base station would have to be aware of all the small differences between each node. We suggest that the node program should be compiled in the same environment whenever possible since it can avoid compatibility issues caused by different compiler versions or different OS versions.

In heterogeneous sensor networks with different hardware, we can employ the same mechanism as just described (i.e., generate different deltas and disseminate deltas using a multicast dissemination protocol). Another possible approach is to employ the

virtual machine technique such that the hardware heterogeneity can be abstracted away. Then our approach can work on top of the VM. The effectiveness of our approach depends on the specific instruction set of the VM.

## 9. CONCLUSION

We present a holistic reprogramming system called R3. The binary differencing algorithm within R3 (R3diff) ensures an optimal result in terms of the delta size under a configurable cost measure. The similarity preserving method within R3 (R3sim) optimizes the binary code format for large similarity with a small metadata overhead. Overall, R3 achieves the smallest delta size compared to other incremental approaches such as Rsync [Jeong and Culler 2004], RMTD [Hu et al. 2009], Zephyr [Panta et al. 2009], Hermes [Panta and Bagchi 2009], and R2 [Dong et al. 2013]. R3's implementation on TelosB/TinyOS is lightweight and efficient. As a future work, we would like to port R3 to other embedded platforms and OSes.

## ACKNOWLEDGMENT

This work is supported by the National Science Foundation of China Grant No. 61202402, the Fundamental Research Funds for the Central Universities, the Research Fund for the Doctoral Program of Higher Education of China (20120101120179), Demonstration of Digital Medical Service and Technology in Destined Region.

## REFERENCES

Wei Dong, Chun Chen, Jiajun Bu, and Chao Huang. 2013. Enabling efficient reprogramming through reduction of executable modules in networked embedded systems. *Ad Hoc Networks* 11, 1 (2013), 473–489.

Wei Dong, Chun Chen, Xue Liu, Jiajun Bu, and Yi Gao. 2011. A lightweight and density-aware reprogramming protocol for wireless sensor networks. *IEEE Transactions on Mobile Computing* 10, 10 (2011), 1403–1415.

Wei Dong, Yunhao Liu, Chun Chen, Jiajun Bu, Chao Huang, and Zhiwei Zhao. 2013. R2: Incremental reprogramming using relocatable code in networked embedded systems. *IEEE Transactions on Computers* 62, 9 (2013), 1837–1849.

Wei Dong, Yunhao Liu, Chao Wang, Xue Liu, Chun Chen, and Jiajun Bu. 2011. Link quality aware code dissemination in wireless sensor networks. In *Proceedings of IEEE ICNP*. 89–98.

Wei Dong, Yunhao Liu, Xiaofan Wu, Lin Gu, and Chun Chen. 2010. Elon: Enabling efficient and long-term reprogramming for wireless sensor networks. In *Proceedings of ACM SIGMETRICS*. 49–60.

Wei Dong, Biyuan Mo, Chao Huang, Yunhao Liu, and Chun Chen. 2013. R3: Optimizing relocatable code for efficient reprogramming in networked embedded systems. In *Proceedings of IEEE INFOCOM*. 315–319.

Adam Dunkels, Niclas Finne, Joakim Eriksson, and Thiemo Voigt. 2006. Run-time dynamic linking for reprogramming wireless sensor networks. In *Proceedings of ACM SenSys*. 15–28.

Adam Dunkels, Björn Grönvall, and Thiemo Voigt. 2004. Contiki—a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of IEEE EmNets*.

Yi Gao, Jiajun Bu, Wei Dong, Chun Chen, Lei Rao, and Xue Liu. 2013. Exploiting concurrency for efficient dissemination in wireless sensor networks. *IEEE Transactions on Parallel and Distributed Systems* 24, 4 (2013), 691–700.

Andrew Hagedorn, David Starobinski, and Ari Trachtenberg. 2008. Rateless Deluge: Over-the-air programming of wireless sensor networks using random linear codes. In *Proceedings of ACM/IEEE IPSN*. 457–466.

Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava. 2005. A dynamic operating system for sensor nodes. In *Proceedings of ACM MobiSys*. 163–176.

Jingtong Hu, Chun Jason Xue, and Yi He. 2009. Reprogramming with minimal transferred data on wireless sensor network. In *Proceedings of IEEE MASS*. 160–167.

Leijun Huang and Sanjeev Setia. 2008. CORD: Energy-efficient reliable bulk data dissemination in sensor networks. In *Proceedings of IEEE INFOCOM*. 574–582.

Jonathan W. Hui and David Culler. 2004. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of ACM SenSys*. 81–94.

Jaein Jeong and David Culler. 2004. Incremental network programming for wireless sensors. In *Proceedings of IEEE SECON*. 25–33.

Joel Koshy and Raju Pandey. 2005a. Remote incremental linking for energy-efficient reprogramming of Sensor Networks. In *Proceedings of EWSN*. 354–365.

Joel Koshy and Raju Pandey. 2005b. VM*: Synthesizing scalable runtime environments for sensor networks. In *Proceedings of ACM SenSys*. 243–254.

Sandeep S. Kulkarni and Limin Wang. 2005. MNP: Multihop network reprogramming service for sensor networks. In *Proceedings of IEEE ICDCS*. 7–16.

John R. Levine. 2000. *Linkers and Loaders*. Morgan Kaufmann.

Philip Levis and David Culler. 2002. Maté: A tiny virtual machine for sensor networks. In *Proceedings of ACM ASPLOS*. 85–95.

Philip Levis, Neil Patel, David Culler, and Scott Shenker. 2004. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proceedings of USENIX NSDI*. 15–28.

Liang, Chieh-Jan Mike, and Andreas Terzis. 2008. Koala: Ultra-low power data retrieval in wireless sensor networks. In *Proceedings of ACM/IEEE IPSN*. 421–432.

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

Lufeng Mo, Yuan He, Yunhao Liu, Jizhong Zhao, Shaojie Tang, Xiang-Yang Li, and Guojun Dai. 2009. Canopy closure estimates with GreenOrbs: Sustainable sensing in the forest. In *Proceedings of ACM SenSys*. 99–112.

Vinayak Naik, Anish Arora, Prasun Sinha, and Hongwei Zhang. 2005. Sprinkler: A reliable and energy efficient data dissemination service for wireless embedded devices. In *Proceedings of IEEE RTSS*. 277–286.

Rajesh K. Panta and Saurabh Bagchi. 2009. Hermes: Fast and energy efficient incremental code updates for wireless sensor networks. In *Proceedings of IEEE INFOCOM*. 639–647.

Rajesh Krishna Panta, Saurabh Bagchi, and Samuel P. Midkiff. 2009. Zephyr: Efficient incremental reprogramming of sensor nodes using function call indirections and difference computation. In *Proceedings of USENIX Annual Technical Conference*.

Rajesh Krishna Panta, Issa Khalil, and Saurabh Bagchi. 2007. Stream: Low overhead wireless reprogramming for sensor networks. In *Proceedings of IEEE INFOCOM*. 928–936.

Joseph Polastre, Robert Szewczyk, and David Culler. 2005. Telos: Enabling ultra-low power wireless research. In *Proceedings of ACM/IEEE IPSN*. 364–369.

Dutta Prabal, Hui Jonathan, Jeong Jaein, Kim Sukun, Sharp Cory, Taneja Jay, Tolle Gilman, Whitehouse Kamin, and Culler David. 2006. Trio: Enabling sustainable and scalable outdoor wireless sensor network deployments. In *Proceedings of ACM/IEEE IPSN*. 407–415.

Michele Rossi, Giovanni Zanca, Luca Stabellini, Riccardo Crepaldi, Albert F. Harris III, and Michele Zorzi. 2008. SYNAPSE: A network reprogramming protocol for wireless sensor networks using Fountain codes. In *Proceedings of IEEE SECON*. 188–196.

C. Sadler and M. Martonosi. 2006. Data compression algorithms for energy-constrained devices in delay tolerant networks. In *Proceedings of ACM SenSys*. 265–278.

Chen Shu, Huang Yan, and Chengyang Zhang. 2008. Toward a real and remote wireless sensor network testbed. In *Proceedings of International Conference on Wireless Algorithms, Systems, and Applications (WASA)*. 385–396.

TIS Committee. 1995. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2*. Retrieved from http://refspecs.freestandards.org/elf/elf.pdf.

Gilman Tolle and David Culler. 2005. Design of an application-cooperative management system for wireless sensor networks. In *Proceedings of EWSN*. 121–132.

Nicolas Tsiftes, Adam Dunkels, and Thiemo Voigt. 2008. Efficient sensor network reprogramming through compression of executable modules. In *Proceedings of IEEE SECON*. 359–367.

Received June 27, 2013; revised October 29, 2013 and January 20, 2014; accepted February 11, 2014