R2: Incremental Reprogramming Using Relocatable Code in Networked Embedded Systems

Wei Dong, *Member*, *IEEE*, Yunhao Liu, *Senior Member*, *IEEE*, Chun Chen, *Member*, *IEEE*, Jiajun Bu, *Member*, *IEEE*, Chao Huang, and Zhiwei Zhao, *Student Member*, *IEEE*

Abstract—We present R2, an incremental reprogramming approach using relocatable code, to improve program similarity for efficient incremental reprogramming in networked embedded systems. R2 achieves a higher degree of similarity than existing approaches by mitigating effects of both function shifts and data shifts. R2 adopts a content-aware differencing algorithm to generate small delta files for efficient dissemination. Besides, it makes efficient use of memory and does not degrade program quality. We implement R2 based on TinyOS 2.1 and demonstrate its advantages through detailed analysis of TinyOS examples. We also present case studies on the software programs of a large-scale sensor system—GreenOrbs. Results show that R2 reduces the dissemination cost by approximately 65 percent compared to state-of-the-art network reprogramming approach—Deluge, and reduces the dissemination cost by approximately 20 percent compared to Zephyr and Hermes—the latest works on incremental reprogramming.

Index Terms—Wireless sensor networks, reprogramming, relocatable code

1 Introduction

Recent advances in microelectronic mechanical systems and wireless communication technologies have fostered the rapid development of networked embedded systems like wireless sensor networks [1], [2]. Despite numerous efforts on the algorithmic research and systematic engineering, management and maintenance tasks of complex systems remain challenging. Enabling the networked system reprogrammable over the air is widely deemed a crucial technology for addressing such challenges [3].

System software needs to be regularly changed in many self-organizing systems for a variety of reasons—fixing bugs, changing network functionality, tuning system parameters, and so on. In our recent efforts in deploying a large-scale and long-term sensor network system—GreenOrbs [4], [5], the importance of network reprogramming cannot be emphasized too much: Network reprogramming significantly reduces the manual efforts involved in traditional ways of collecting all nodes back, attaching to computers to "burn" new codes, and redeploying all nodes in the field.

In the network reprogramming process, the new program code needs to be disseminated to all nodes in the network. It is highly desirable to reduce the transferred code size to shorten the reprogramming time, reduce the transmission overhead, and reduce the energy consumption. In this work, we propose an incremental reprogramming approach to effectively reduce the transferred code size by exploiting the similarity between two program versions.

The key idea to improve program similarity is to keep multiple references to the same symbol unchanged when the program changes. For example, consider a function that is referenced at multiple locations in the program. In the standard binary generation process, if the function is allocated to a different address in the new version, all the references to this function changes, resulting in low similarity between two program versions. Similar problems exist for global data variables.

Koshy and Pandey [6] propose a mechanism to 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 [6]. Zephyr [7] mitigates the effects of function shifts by using a function indirection table. All function calls in the program are indirected via fixed table slots to their actual implementations. The approach does not handle instructions other than `call`, which decreases the similarity that can be achieved. Moreover, it decreases the execution efficiency because of indirections.

Hermes [8] mitigates the effects of *data shifts* by first pinning variables to the same locations by source-level modifications, and then adopt two different approaches for handling data shifts. For the Von-Neumann-based TelosB nodes, it allocates `.bss` variables to the program flash. Such an approach leads to decreased execution

- *W. Dong, C. Chen, J. Bu, C. Huang, and Z. Zhao are with the Zhejiang Key Lab of Service Robot, College of Computer Science, Zhejiang University, Zheda Road 38, Hangzhou 310027, China. E-mail: chenc@zju.edu.cn.*
- *Y. Liu is with the School of Software and TNLIST, Tsinghua University, Beijing, China 100084. E-mail: yunhao@greenorbs.com.*

Manuscript received 11 Dec. 2011; revised 14 June 2012; accepted 17 June 2012; published online 25 June 2012.
Recommended for acceptance by P. Bellavista.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2011-12-0945.
Digital Object Identifier no. 10.1109/TC.2012.161.

efficiency as flash I/O operations incur more overhead than RAM operations. More importantly, program flash has a limited number of I/Os, making the approach inappropriate for memory-intensive and long-term applications. For the Harvard-based MicaZ nodes, it leaves a slop region (of 10) 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.

To address the above limitations, we propose a unified approach called R2 to mitigate both effects of *function shifts* and *data shifts* by using *relocatable code*. Our approach obtains a higher degree of similarity by keeping all references in the instructions the same in both versions of programs. Moreover, it makes efficient use of memory and does not degrade program quality.

Relocatable code is a well-established technique for dynamic linking and loading [9]. To our knowledge, it is novel to employ this technique to increase code similarity for incremental reprogramming. A direct use of relocatable code, however, would cause a large code size for reprogramming. To address this issue, we adopt several techniques. First, we reduce the entry size in the relocation table. Second, we carefully organize the relocation entries according to their types (i.e., function or data). This organization makes the difference of the relocation table even smaller. Third, we propose a content-aware differencing algorithm to further reduce the difference of the relocation table (as detailed in Section 3.3.3), exploiting special patterns exhibited in the relocation table. These three techniques are very important to reduce the overhead of relocation table.

We integrate the similarity improvement approach (R2sim) and the differencing algorithm (R2diff) into an incremental reprogramming system based on TinyOS 2.1 [10]. We call it R2, incremental reprogramming using relocatable code. R2 consists of four steps in achieving incremental reprogramming. First, we modify the standard code generation process in TinyOS for generating relocatable code. Second, we devise a delta file format and apply the content-aware differencing algorithm to generate a small delta for dissemination. Third, we incorporate a code dissemination protocol based on Deluge [11] and Stream [12] for code dissemination. Finally, we implement a delta interpreter and a dynamic loader for reconstructing and executing the new program.

We examine R2's performance through analysis of TinyOS examples as well as results obtained from software programs for our 330-node sensor system—GreenOrbs. Results show that 1) R2 improves the program similarity compared to existing approaches; 2) R2 preserves program quality in terms of memory efficiency and execution efficiency; 3) integrated with the differencing algorithm, R2 reduces the delta size further, achieving approximately 20 percent reduction compared to Hermes [8].

The contributions of this work are summarized as follows:

- *A unified approach.* R2 mitigates both effects of function shifts and data shifts by employing the concept of relocatable code, so that it obtains a higher degree of similarity than existing approaches. Moreover, it makes efficient use of memory and does not decrease program quality. Although relocatable code has been used in traditional computing systems for dynamic linking and loading for a long history, the use of relocatable code in incremental reprogramming for networked embedded systems is novel.
- *An optimized differencing algorithm.* Compared to block-level differencing algorithms like Rsync [13], R2diff results in smaller delta size because it performs comparisons at the byte level. Compared to existing byte level differencing algorithms like RMTD [14], R2diff incorporates a novel content-aware differencing algorithm to reduce the difference of the relocation table, resulting in 40-70 percent reductions for the relocation table.
- *A holistic system.* R2 integrates the similarity improvement approach and the optimal differencing algorithm into a reprogramming system, based on TinyOS 2.1. It uses a persistent dissemination service to further improve the reprogramming reliability.

The rest of this paper is organized as follows: Section 2 describes the motivation of our work. Section 3 presents the design of R2. Section 4 shows the evaluation results. Section 5 describes the related work. Finally, we conclude this paper in Section 6.

2 Motivation

We first show the importance of incremental reprogramming for networked embedded systems (Section 2.1), and then discuss the benefits of improving the program similarity (Section 2.2). We also summarize the major limitations of existing approaches (Section 2.3).

2.1 Why Incremental Reprogramming?

Networked embedded systems, such as wireless sensor networks, need to run software program to fulfill the application requirements. It is often difficult for developers to guarantee the networked nodes to behave as expected and even meet the future needs. Indeed, it is impossible to anticipate and test the software under all circumstances, especially when a large-scale system is deployed in the wild.

To illustrate the difficulty, let us look at a real example we experienced when incorporating state-of-the-art time synchronization protocol—FTSP [15], into our GreenOrbs system. During GreenOrbs' 10-month deployment in the Zhejiang Forestry University's woodland, we often observe that some nodes frequently lose synchronization with the rest of the network. These nodes suppose to stay awake to be synchronized, resulting in high energy consumption. After many rounds of careful detections, we realized that the `PacketTimeStamp` bug in the CC2420 radio chip caused this problem. We have printed the local time stamp read by the driver and the local time stamp read by ourselves when the packet is signaled to an event handler. Ideally, both time stamps should be roughly the same, i.e., the ratio between them should be approximately 1. However, some ratios are unexpectedly high, clearly illustrating the error readings in the driver. Worse, nodes propagate those errors to others, causing instability of the entire network. Since those errors simply cannot be expected and can happen at any level of the software, network reprogramming for fixing those bugs are essential to pull the system back to the correct state.

Unfortunately, state-of-arts reprogramming approaches, such as Deluge [11] and Stream [12], are far from efficient in disseminating the new code. For example, the GreenOrbs program code consumes approximately 45 Kbyte, approaching the limit of TelosB's total size of program flash. In other real-world systems, for example, VigilNet [16], the program code is also quite large. A large transferred code size would result in a long dissemination time, high transmission overhead, and high energy consumption. Hence, it is highly desirable to reduce the transferred code size by disseminating the deltas. For example, in our initial attempt in reducing the transferred code size by using the RMTD incremental reprogramming approach [14], we can reduce the transferred code size by nearly 50 percent, significantly reducing the transmission overhead during network reprogramming.

2.2 Why Improving the Program Similarity?

Without improving program similarity between two program versions, the generated delta size can sometimes disappointedly large than expected.

Consider a software change case of adding a few lines to a function in a program. This will cause all functions after the modified one shift to different addresses in the new program. It is not uncommon that there can be multiple references to a single function. The shift of a function can potentially cause huge differences in all the relevant references between two program versions.

Consider another software change case of adding a global data variable. This will cause all data variables after this newly added one shift to different addresses in the new program. Similarly, since there can be multiple references to a single data variable, the shift of a data variable can potentially cause huge differences in all relevant references between two program versions.

These two examples illustrate the discrepancy in the delta size expected by the system developer and that actually produced by the existing tools—the application-level modifications are sufficiently small while the differences between two program versions are huge.

Obviously, keeping all the references the same between two program versions can bridge the gap, increasing the program similarity and reducing the delta size. Nevertheless, various technical challenges need to be addressed for achieving the maximum similarity while ensuring a small delta for dissemination and preserving a high program quality for execution.

2.3 Limitations of Existing Approaches

We summarize pros and cons of existing approaches in the following sections.

2.3.1 Difference Computation

Differential compression is a well-studied topic in traditional P2P applications, file synchronization, version control systems, and software distribution over the Internet [17], [18], [19], [20]. Traditional approaches adopt block-based techniques to reduce the time and space complexity of generating the delta. Although these approaches are effective in handling large files in resource-abundant computer systems, they are not optimal in terms of the delta size for resource-constrained microembedded systems such as sensor nodes.

Jeong and Culler [13] propose a modified Rsync algorithm [21] for efficient incremental reprogramming for sensor nodes. Block-based comparison in Rsync cannot find common segments (CS) with length smaller than the block size. The optimized Rsync algorithm proposed in [7] employs various techniques to reduce the delta size, for example, combining multiple contiguous blocks to the longest one. However, it does not guarantee the smallest delta size.

Having a byte-level differencing algorithm is possible for microembedded systems because of two reasons. First, the delta computation process is performed on resource-abundant computers, rather than microsensor nodes. Second, the program code size for microembedded system is usually small, for example, limited to 48 Kbyte for TelosB nodes and 128 Kbyte for MicaZ nodes.

The RMTD algorithm proposed in [14] is an optimal byte-level differencing algorithm for incremental reprogramming. While the algorithm is effective in handling native code, it is, however, not optimized in handling relocatable code because of the overhead in metadata. It can sometimes generate a large delta file without exploiting special patterns exhibited in the metadata.

2.3.2 Similarity versus Program Quality

Some approaches have been proposed to improve the program similarity [6], [7], [8]. The key idea in those approaches is to keep the references unchanged when the program changes. Separate approaches for addressing function shifts and data shifts have been proposed. Approaches mitigating function shifts include the slop region approach [6], Zephyr's indirection table approach [7]. Approaches mitigating data shifts include Hermes' approach [8].

These approaches are not unified in handling function shifts and data shifts. More importantly, they suffer from the following limitations:

- *Limited similarity*. For example, the slop region approach [6] and Hermes (for MicaZ) [8] do not address the case when functions or data grow out of the slop regions. Zephyr [7] and Hermes [8] do not handle references in instructions other than the `call` instruction.
- *Memory inefficiency*. For example, the slop regions in [6] and Hermes (for MicaZ) [8] consume extra memory space. They also cause memory fragmentation when some functions or data are deleted in the new version. The function indirection table used in Zephyr [7] and Hermes [8] occupies additional program address space.
- *Execution inefficiency*. For example, Zephyr [7] has the extra overhead of indirections. Hermes (for TelosB) [8] has the extra overhead in accessing the .bss variables (since it stores .bss variables in the program flash).

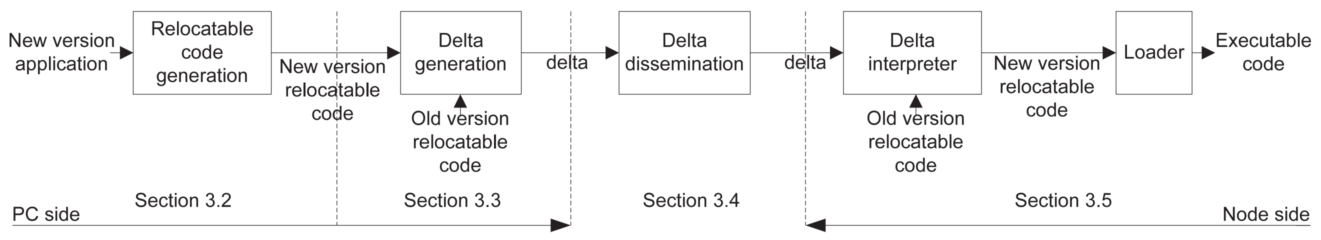


Fig. 1. An overview of the R2 incremental reprogramming system.

3 Design

We first give an architecture overview, including overall design goals and steps involved in network incremental reprogramming, and then describe R2 in four specific steps.

3.1 Overview

Our overall design goal is to develop an efficient incremental reprogramming system—R2, that achieves small transmission overhead. As such, we need to propose effective approaches to optimize the transferred delta size, and design corresponding PC-side utilities and node-side system software. Our approach should address limitations of existing approaches summarized in Section 2.3 and have the following features:

- *High degree of similarity*. R2 should handle a diversity of reference instructions, for example, `call`, `mov`, `br`, and so on.
- *Unified approach*. R2 should unify approaches for handling function shifts and data shifts.
- *High program quality*. R2 should preserve the high program quality in terms of runtime execution efficiency and memory efficiency.
- *Small delta size*. R2 should optimize the overall delta size to improve the reprogramming efficiency in terms of reprogramming time and transmission overhead.
- *Lightweight for implementation*. The implementation of R2 should meet the resource constraints of microembedded systems.

Fig. 1 gives an overview of the R2 reprogramming system. To reprogram the network nodes, we generate the relocatable code for the new application using R2's relocatable code generation method, R2sim (Section 3.2). Our differencing algorithm, R2diff, compares the newly generated relocatable code against the old version and generates a small delta for dissemination (Section 3.3). Then, the dissemination protocol disseminates the delta to all nodes in the network (Section 3.4). Finally, the delta interpreter on the nodes reconstructs the new program which is then loaded onto the program flash for execution (Section 3.5).

3.2 R2sim: Generating Relocatable Code

The intention of applying relocatable code is to make all references to symbols (i.e., either functions or global data variables) the same when the program changes. By filling the absolute addresses in those reference instructions to a predefined value (e.g., zero) in all program versions, we keep those reference instructions the same when the functionality of the program changes. To make these reference instructions execute correctly at runtime, we must generate additional metadata, i.e., relocation entries, for the node-side loader to perform load-time modifications. In general, a relocation entry contains the following information: 1) the memory location to apply the modification, and 2) the correct target address for the reference instruction. While relocatable code is a common technique for traditional dynamic linking and loading [9], we employ it to improve the program similarity. The existence of the relocation entries allows the decoupling of symbol reference and symbol definition: We keep all the references unchanged while the actual target addresses can be different between two program versions.

We modify the code generation process of TinyOS/nesC to generate relocatable code. First, we let the linker to generate relocation entries in the final executable file (ELF). Then, we parse the ELF file, and fill corresponding target addresses in all reference instructions to zero. We combine the code and the relocation table into a binary file for delta generation (Section 3.3).

We generate a relocation entry for each reference. In the standard ELF, the relocation entry is of the following type:

```
typedef struct {
  Elf32_Addr r_offset;
  Elf32_Word r_info;
  Elf32_Word r_addend;
} Elf32_Rela;
```

where `r_offset` tells the memory location to apply the relocation. `r_info`, together with `r_addend`, tells information on how to apply the relocation (e.g., what is the target address). When a reference instruction refers to a symbol, `r_info` can be used to locate the symbol's address, and `r_addend` is added to the symbol's address to get the final target address.

The above relocation entry consumes 12 bytes. For microembedded systems, it is important to compress the entry size. To this end, we compress R2's relocation entry as follows:

```
typedef struct {
  uint16_t r_offset;
  uint16_t r_addr;
} rela_t;
```

where `r_offset` tells the memory location to apply the relocation, and `r_addr` is the final target address. We can obtain the value of `r_addr` after we have parsed the relocatable ELF file. Each compressed relocation entry consumes only 4 bytes, 66 percent reduction compared to the entry in standard ELF.

We only use a compiler flag to let the compiler generate relocation entries for reference instructions. The use of this flag will not affect the generation of the binary instructions. To confirm this, we conduct two experiments:

1. In the first experiment, we count the total number of functions (inline+noninline) in the source file (app.c in TinyOS) as well as the number of functions (noninline) in the assembly file. For the two compilations (i.e., TinyOS standard compilation and R2's compilation with relocatable code), we find that the number of inline functions keeps the same for all the benchmarks investigated in this paper.
2. In the second experiment, we use our binary differencing tool (R2diff) to compare the code section from two compilations. We note that the use of R2sim will inflate the reference addresses to zero, causing unnecessary difference. These addresses will be corrected at load time before execution. Therefore, we compare two code sections without the use of R2sim. We find that both the code sections are exactly the same for all the investigated benchmarks investigated in this paper. Hence, the additional generation of the relocation table does not affect the optimization of TinyOS programs.

The advantages of utilizing relocatable code are summarized as follows:

- It is supported by the compiler and linker.
- It is a unified approach in handling function shifts and data shifts. It can handle more types of reference instructions than Zephyr and Hermes' approach [7], [8], achieving a higher degree of program similarity compared to existing approaches.
- It preserves a high program quality in terms of runtime execution efficiency and memory efficiency. In particular,
    - it does incur indirection cost;
    - no slop regions are needed and data can be compacted when some variables are deleted in the new program.

It is worth mentioning the difference of our design and the indirection approach of Zephyr [7]. In Zephyr, each indirection table slot consumes 6 bytes (a `call` instruction consumes 4 bytes and a `ret` instruction consumes 2 bytes). However, the number of indirection table slots is less than the number of relocation entries because it handles less types of reference instructions. The tradeoff is that it results in a lower degree of similarity that increases the delta size. Indeed, the metadata overhead of the relocation table can be large for complex programs. Existing differencing algorithms may fail to generate a small delta because of the metadata overhead. To address this issue, we need to redevise the differencing algorithm to exploit special patterns exhibited in the metadata to reduce the delta size. We will present design details of the differencing algorithm in the following section.

3.3 R2diff: Generating the Delta

We now present our differencing algorithm for generating the delta. For the code, we adopt the byte-level RMTD algorithm that has $O(n^3)$ time complexity and $O(n^2)$ space complexity [14]. For the metadata of relocation entries, we exploit special patterns to perform content-aware comparisons. We first give the basic commands in the delta script (Section 3.3.1), and then we present the differencing algorithm in Sections 3.3.2 and 3.3.3, for the code and metadata, respectively.

3.3.1 Cost Measure

We introduce two basic commands in the delta script. The first command is the `ADD` command of the following form:

```
ADD <n> <BYTE1 .. BYTEn>
```

where `n` is the number of bytes added, followed by `n` bytes. We assume the add command costs n bytes.

The second command is the `COPY` command of the following form:

```
COPY <n> <old_addr>
```

where `n` is the length of bytes copied, starting from `old_addr` in the code program. Note that the starting address in the new program is not needed because we construct the new code in sequential order. We assume the copy command costs $\beta = 5$ bytes.

3.3.2 Byte-Level Comparison

We first introduce the RMTD algorithm that is a byte-level differencing algorithm with minimum transferred delta [14]. It is used for the code section. We then use an simple example to illustrate why R2 can reduce the overall delta size.

The basic idea of RMTD is described as follows:

1. It uses a two-dimensional table to record whether each pair of bytes in the two code versions is the same. Then, the RMTD algorithm searches in the two-dimensional table for CS between two code versions. It stores all common segments in a linked list.
2. Let opt_i be the minimum transferred delta size to construct the first i bytes of the new program. We can get
    a. $opt_0 = 0$.
    b. $opt_i \geq opt_{i-1}$, 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 minimum delta size used to construct $i-1$ bytes should be no smaller than opt_i.

RMTD uses a dynamic programming approach. The minimum bytes used to construct i bytes in the new program is computed according to (1), given at the bottom of the page.

$$opt_i = \begin{cases} 0 & i = 0 \\ opt_{i-1} + 1 & i > 0, k = \texttt{findK(i)} = \infty, \\ & \text{i.e., } i \text{ is not covered by any CS} \\ \min(opt_k + \beta, \\ \quad opt_{i-1} + 1) & i > 0, k = \texttt{findK(i)} \neq \infty. \end{cases} \quad (1)$$

We explain this equation as follows: We use a procedure `findK` to find the smallest index k where $[k, i]$ matches a segment in the old program. If there is no match, `findK` will return the maximum integer value. If the current byte is

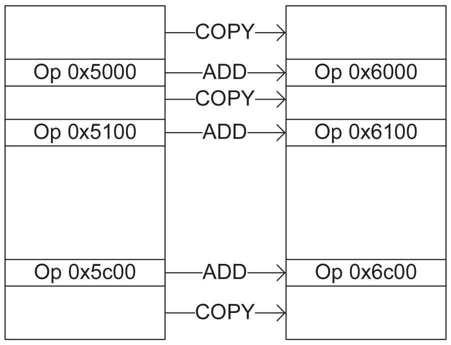


Fig. 2. An example.

not covered by any CS, we must add the byte. If the current byte is covered by CSs, we find the smallest index k. Note that any other copy scheme starting from $k' > k$ will not be better because $opt_{k'} \geq opt_k$.

To see why R2 can reduce the overall delta size, let us consider an example shown in Fig. 2. Consider two code segments with n reference instructions changed in the target addresses. The native incremental approach (i.e., without preserving similarity) 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 $(n+1) \times 5 + n \times 2 = 7n + 5$. Suppose there are k call instructions among the total n reference instructions, and m unique symbols are referenced for all the n reference instructions. Zephyr/Hermes need $7(n-k)+5$ bytes since only address fields in the $(n-k)$ noncall instructions need to be added. The metadata overhead consumes at most $2m$ bytes (the metadata is also transferred by a delta so the actual transferred overhead can be smaller than $2m$). The delta size of Zephyr/Hermes is $10(n-k)+5+M[2m]$ where $M[x]$ denotes the delta size for the x-byte metadata ($0 \leq M[x] \leq x$). With R2, we only need one COPY command, which consumes 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 $5 + M[4n]$.

We summarize the results in Table 1. We investigate all benchmarks in this paper and find that there are about 40 percent call instructions among all reference instructions, i.e., $k \approx 0.4n$. Therefore, it can be seen from Table 1 that R2 results in the smallest delta size for sufficiently large n.

3.3.3 Optimization for the Metadata

We enhance the optimal differencing algorithm described in the previous section for optimizing the metadata. First, the algorithm performs differential comparison at the granularity of each entry because changes to the relocation entries are at this granularity. Second, the relocation entries

TABLE 1
Delta Size for Example Shown in Fig. 2

Approach	Delta size
Native	7n+5
Zephyr/Hermes	7(n-k)+5+M[2m]
R2	5+M[4n]

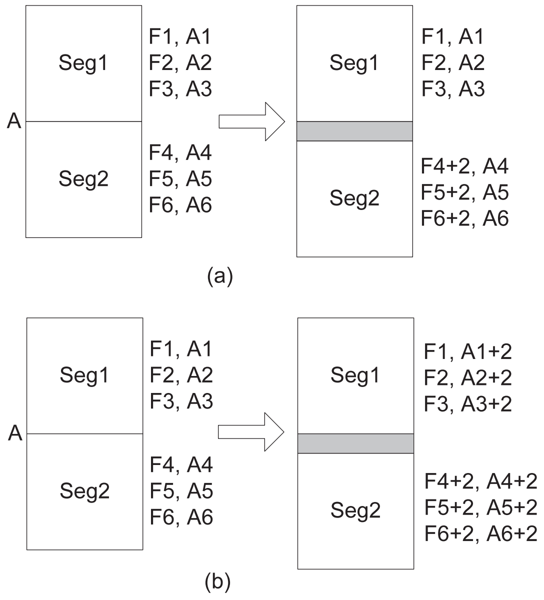


Fig. 3. An illustrative example. (F_i, A_i) denotes the relocation entry. (a) $A_i < A$. (b) $A_i > A + 2$.

are organized according to their types (e.g., function or data). Third, the algorithm performs content-aware comparisons, exploiting special patterns in the metadata for reducing the delta size.

We consider a software change case of adding an instruction of 2 bytes. Fig. 3 gives a graphical illustration. The instruction is inserted at address A. Assume there are six relocation entries for the old code, i.e., (F_i, A_i), $1 \leq i \leq 6$, where F_i tells at which address relocation should be performed, and A_i is the actual target address that should be filled at address F_i.

We consider two cases here: 1) All symbol addresses are lower than A where the change occurs, i.e., $A_i < A$, $1 \leq i \leq 6$. In this case, the relocation entries for `Seg1` will not change. However, the relocation entries for `Seg2` will be changed to $(F_i + 2, A_i)$, $4 \leq i \leq 6$. This is because the locations to perform relocation shift by 2 bytes. 2) All symbol addresses are higher than $A + 2$, i.e., $A_i > A + 2$, $1 \leq i \leq 6$. In this case, the relocation entries for `Seg1` will be changed to $(F_i, A_i + 2)$, $1 \leq i \leq 3$. The relocation entries for `Seg2` will be changed to $(F_i + 2, A_i + 2)$, $4 \leq i \leq 6$.

To summarize, we find it is common that

- The values of F_i shift for a consecutive of entries, while the values of A_i keep the same as in the old version.
- The values of A_i shift for a consecutive of entries, while the values of F_i keep the same as in the old version.
- Both the values of F_i and A_i shift for some constants.

We provide three extra delta commands to optimize for the metadata

```
CE    <n>  <old_addr>
CEX   <n>  <old_addr>  <x_off>
CEY   <n>  <old_addr>  <y_off>
CEXY  <n>  <old_addr>  <x_off>  <y_off>
```

where n is the number of relocation entries. The first command copies the specified number of relocation entries. The remaining commands copies the entries by adding the specified constants to the corresponding fields.

To see the benefits, we look back to the example shown in Fig. 3. In case (a), we can encode the change for relocation entries as two delta commands, i.e., `CE 3 addr1`, `CEX 3 addr2 2`, where `addr1` and `addr2` are addresses of (F_1, A_1) and (F_4, A_4) in the old code. In case (b), we can also encode the change for the relocation entries as two delta commands, i.e., `CEY 3 addr1 2`, `CEXY 3 addr2 2 2`. Without these optimizations, we have to explicitly add the changed bytes, resulting larger delta size.

To implement the above optimizations, we extend the RMTD algorithm by recording the difference for a pair of entries in the two-dimensional table. Hence, we can find common segments with constant offsets.

After obtaining the delta according to the above commands, there are still chances in reducing the number of added bytes specified in the delta. We note that a symbol can be referenced by multiple instructions, i.e., multiple entries may have the same target address. If these entries appear in the added bytes, we provide an additional metacommand for compression (here, the subscript denotes the number bytes consumed by each field)

$$\texttt{AE1}_1\ \texttt{<n>}_2\ \texttt{<r\_addr>}_2\ \texttt{<r\_off1..r\_offn>}_{2n}$$

With this metacommand, we can reduce $4n$ bytes to $5 + 2n$ bytes.

It is also worth noting that when interpreting this command, the entries may be decoded out of order. To preserve the order, we need to sort the entries before storing them onto the external flash.

3.4 Disseminating the Delta

The delta dissemination protocol is similar to Deluge [12]. Deluge works as follows: A node advertises the program version using a Trickle timer [22]: The advertisement interval decreases to the minimum when a new program version is found while doubles to the maximum when the network is steady. When the advertisement of a new program 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-sized pages to enable pipelining and employs NACK to achieve reliability.

To make the node software reprogrammable for multiple times, Deluge needs to be wired into the application so that each node can always respond to reprogramming commands. We find that, however, the Deluge code in TinyOS 2.1 consumes more than 30 Kbyte. It cannot be wired into our GreenOrbs application because the GreenOrbs program already consumes approximately 45 Kbyte, approaching the maximum allowable program size of 48 Kbyte on TelosB nodes.

To address this issue, we use the Stream [12] 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.

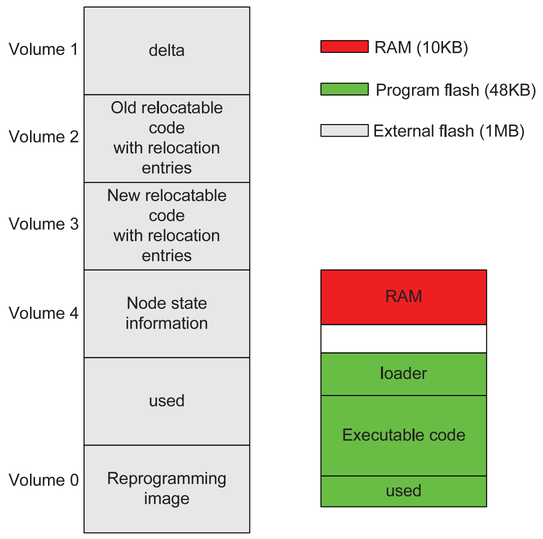


Fig. 4. Memory layout with R2.

While this approach can address the problem of space limitations, it causes reliability issues. The problem we encountered in applying the Stream approach in network reprogramming is that it is difficult to let all nodes in the same state with a single broadcast, especially in networks with unreliable links. To address this issue, we have used a separate dissemination service for node states. This service is based on Drip [23] for disseminating small data items and works as follows:

1. Nodes exchange state information for multiple rounds, ensuring 100 percent reliability.
2. A node records
    a. its running state (i.e., reprogramming state or application state) with a state version number. A node should switch to the state with a newer version number.
    b. the application program version. When a new program version is found, a node should request for obtaining the new program.
3. Information of 2a and 2b is persistent across node reboots. We use a separate volume in the external flash for storing and restoring this information.

3.5 Reconstructing Executable Code

Fig. 4 shows the memory layout with R2. We store the program code and the delta script in the external flash during code dissemination. The external flash on TelosB nodes has 1-Mbyte space. R2 consumes five volumes, each of which consumes 64-Kbyte space. Volume 0 stores the reprogramming image. Volume 1 stores the delta script received. Volumes 2 and 3 store program code (including relocatable code and metadata) of two versions. Volume 4 stores the state information that should be persistent across node reboots.

There are two steps for reconstructing the new executable code. First, the delta interpreter included in the reprogramming image reconstructs the new relocatable code according to the old version and the delta received. Second, the dynamic loader loads the reconstructed code onto the program flash. During the loading process, it performs relocating according to information in relocation entries. Finally, nodes can execute the new executable code by forcing a reboot.

4 Evaluation

This section evaluates the effectiveness of our design. Section 4.1 introduces our evaluation methodology. Section 4.2 shows the effectiveness of R2sim and R2diff using TinyOS microbenchmarks. Section 4.3 presents case studies (including real-world software change cases in the development of the GreenOrbs application) to illustrate the benefits of R2 in improving reprogramming efficiency. Section 4.4 shows the reconstruction overhead on the nodes. Section 4.5 shows the dissemination time and energy consumption in a 24-node testbed, and Section 4.6 gives a summary of results.

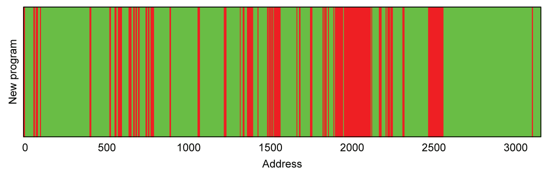
4.1 Methodology

We implement R2 based on TinyOS 2.1 for TelosB nodes. The TelosB node uses the msp430f1611 microcontroller with variable length instructions. It has 10-Kbyte RAM for data and the stack, 48-Kbyte program flash for the program code. It has an additional 1-Mbyte external flash for storing persistent data. Toward implementing R2, we write programs for

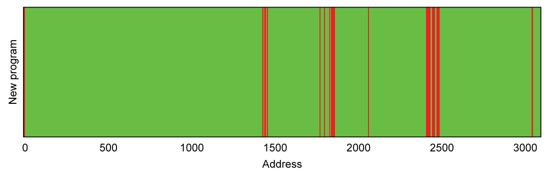
1. R2sim, i.e., the relocatable code generation method on the PC side (in Perl),
2. R2diff, i.e., the differencing algorithm and its optimizations for relocation entries on the PC side (in C),
3. the Deluge/Stream-based dissemination protocol using persistent dissemination service on the node side (in nesC), and
4. the delta interpreter and the dynamic loader on the node side (in nesC).

We have also used the code of Java implementation of Rsync [13] for comparing performance of differencing algorithms. We implement the Zephyr/Hermes [7], [8] approach for comparing the performance of similarity improvement approaches.

We first use TinyOS microbenchmarks to study the performance of R2sim and R2diff. We then study several real-world change cases in the development of GreenOrbs application to examine the effectiveness of R2 in improving the overall reprogramming efficiency. GreenOrbs [4], [5] is a recently deployed sensor network system that aims at achieving long-term kilo-scale surveillance in the vast forest. The current GreenOrbs software program builds on top of the TinyOS 2.1. The GreenOrbs program includes the CTP component [24] for collecting multiple types of sensor data, for example, light, temperature, humidity, to a collection sink. To increase the flexibility, GreenOrbs includes the Drip component [23] for disseminating key system parameters, for example, duty cycle ratio, transmission power, and so on. To achieve energy efficiency, GreenOrbs also includes the FTSP component [15] for enabling synchronous low duty cycling. We also examine R2's overhead in reconstructing the new program code.



(a) Hermes (595 bytes need to be added)



(b) R2 (93 bytes need to be added)

Fig. 5. Similarity comparison of two approaches.

4.2 Microbenchmarks

4.2.1 Evaluation of R2sim

We first look at the `Blink` example in TinyOS 2.1. We change `Blink` to `CntToLeds`.

The number of bytes in the new program that cannot be found in a common segment in the old program must be encoded in the `ADD` command in the delta file. Thus, the number of added bytes measure the degree of program similarity. We compare similarity achieved by three approaches: 1) native: optimal diff without any similarity improvement methods, 2) Hermes: optimal diff with Hermes optimization, 3) R2: optimal diff with R2 optimization. We have found that R2 achieves the highest degree of similarity: It requires 82 added bytes while the native approach requires 478 bytes and Hermes requires 134 bytes. R2 achieves greater similarity than Hermes because it handles more types of reference instructions. For example, we have found that some instructions (other than `call`) can also reference symbol addresses, for example, the branch instruction `br`, the `mov` instruction, interrupt services routines (from address `0xffe0` to address `0xffff`).

R2 achieves higher similarity when data are added to a program. In Hermes, when some data expand the predefined slop region (into the `bss` section), the overlapped `bss` variables are forced to be relocated, causing references to these variables change. To illustrate this, we add 100 bytes initialized data to the `CntToLeds` program, adding one reference to each of the data. In Hermes, half of the total 180 bytes `bss` variables are forced to be relocated. Fig. 5 shows the added bytes at different locations in the new program. We can see that while R2 only slightly decreases the similarity (the number of added bytes increases from 82 to 93 bytes), Hermes decreases the similarity significantly (the number of added bytes increases from 134 to 595 bytes). Hence, R2 is more general in unifying the techniques for handling both functions shifts and data shifts, achieving a higher degree of similarity than existing approaches.

We then perform controlled insertions to an application using the collection and dissemination services. In the first case, we insert code of length 50, 100, 150, 200, 250 bytes at the beginning of the program. Note that we can only

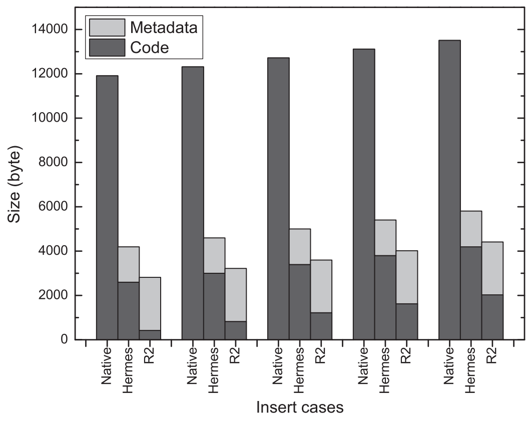


Fig. 6. Comparison of three incremental approaches for the first case.

perform insertions at the source code level since if we insert code at the binary level, we would corrupt the reference instructions in the binary code. This also means that we cannot precisely control the insertion point and the byte-distance between two insertions. In the second case, we perform 10, 20, 30, 40, 50 insertions at manually selected locations, each with a code size of 10 bytes. Figs. 6 and 7 show the results of three incremental approaches, i.e., native, Hermes, and R2. We can see that in the first case, R2 reduces the cost by more than 20 percent while in the second case, the improvement becomes smaller. This is because R2's metadata size becomes larger. However, in real-word software change cases, R2 achieves 20 percent improvement compared to Hermes since the code size reduction is more effective for complex changes as observed in Section 4.3.2.

It is worth mentioning some additional benefits of R2. R2 does not incur Zephyr's runtime execution overhead of function indirections [7]. Function indirection degrades execution efficiency, which is not desired for long-running computation-intensive applications.

R2 makes efficient use of memory. The indirection table slots in Zephyr consume extra program address space. This address space cannot be utilized by the useful program codes on platforms without virtual address space. Hermes' approach in handling data shifts causes extra RAM consumption because of the slop region. If the slop region is too small, data still need to be shifted when new variables are added.

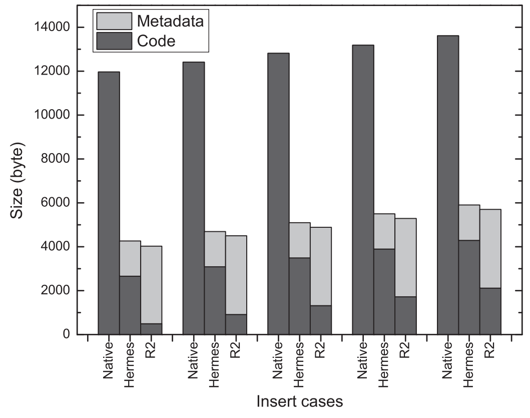


Fig. 7. Comparison of three incremental reprogramming approaches for the second case.

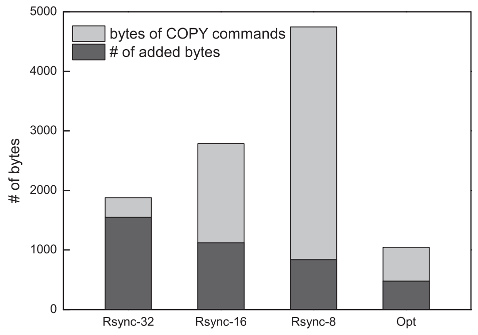


Fig. 8. Evaluation of differencing algorithms.

4.2.2 *Evaluation of R2diff*

In this section, we evaluate the effectiveness of R2diff in generating the delta file.

First, we would like to examine how byte-level comparison improves the performance compared to block-level comparison like Rsync [13].

We change the `Blink` application to `CntToLeds` application. We add a variable and show its last three bits though blinking the leds. We compare the delta sizes generated by Rsync and our optimal differencing algorithm.

Fig. 8 shows the results. The deltas generated by Rsync are of different sizes under different block sizes. With a large block size (e.g., 32 bytes), there is more number of added bytes because the number of common blocks is fewer. With a small block size (e.g., 4 bytes), the `COPY` commands would consume more number of bytes. Our optimal differencing algorithm generates the smallest delta size, with fewest number of added bytes.

Continuing the same change case, we look at the effectiveness of our content-aware differencing algorithm when applied to the relocation entries. The total size of the relocation entries in the new program version consumes 788 bytes. We find that directly applying the optimal differencing algorithm generates a delta with 712 bytes. On the other hand, using content-aware optimization can reduce the size to 481 bytes. Optimizations using metacommands can further reduce the size to 421 bytes. We see that R2's content-aware comparison results in a 41 percent reduction.

For relatively large applications involving data collection and dissemination, we consider the two change cases described in the previous section. The first case inserts 250 bytes to the application and the second case performs 50 insertions, each with 10 bytes.

TABLE 2
Delta of the Metadata (Bytes)

Case	size of reloc	diff	diff+optimization
First case	14,160	7,054	2,316
Second case	14,164	7,146	3,345

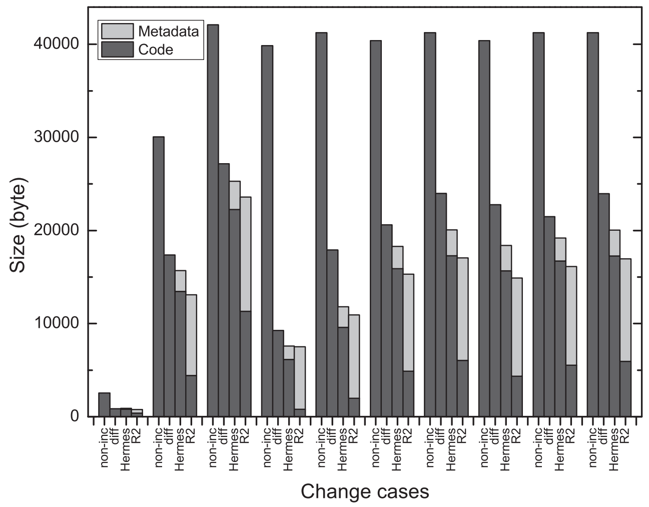


Fig. 9. Delta size comparison of four reprogramming approaches.

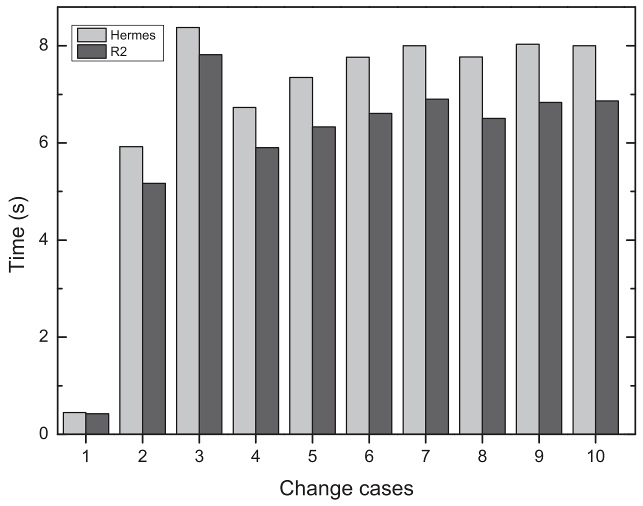


Fig. 10. Reconstruction overhead.

Table 2 shows that 1) performing incremental update on the relocation table can reduce the overhead by approximately 50 percent; 2) performing incremental update with optimizations described in Section 3.3.3 can further reduce the overhead by 50-70 percent. This illustrates that the optimization techniques are very important in reducing the delta size of the metadata, especially for relatively complex applications.

4.3 Case Studies

4.3.1 The Change Cases

We examine the delta size generated by R2 compared to Zephyr/Hermes [7], [8] via 10 change cases described as follows:

- *Case* 1: We change the Blink application to CntToLeds application.
- *Case* 2: We change the TestNetwork application to TestNetworkLpl.
- *Case* 3: We consider the GreenOrbs application of SVN version 168. The base version performs sensing, transmissions, and so on, while the updated version disables sensing and enables local logging on the external flash.
- *Case* 4: We consider the change case of GreenOrbs from version 182 to 183. The updated version fixes a link estimation bug in the 4 bitle component.
- *Case* 5: We consider the change case of GreenOrbs from version 306 to 314. The updated version provides another parameter for reconfiguration.
- *Case* 6: We consider the change case of GreenOrbs from version 168 to 306. The updated version adds more components.
- *Case* 7: We consider the change case of GreenOrbs from version 168 to 314. The updated version adds more components and parameter for reconfiguration.
- *Case* 8: We consider the change case of GreenOrbs from version 182 to 306.
- *Case* 9: We consider the change case of GreenOrbs from version 182 to 314.
- *Case* 10: We consider the change case of GreenOrbs from version 177 to 314.

4.3.2 Overall Delta Size

To examine how R2 improves the efficiency of state-of-arts reprogramming protocols, we evaluate

1. nonincremental reprogramming by disseminating the new program code.
2. the deltas generated without optimizing the similarity.
3. the deltas generated by Hermes.
4. the deltas generated by R2.

Fig. 9 shows the comparison results. We can see that 1) incremental reprogramming is effective in reducing the transferred code sizes, resulting in 30-50 percent reductions in the number of transferred bytes. 2) R2 significantly improves the similarity compared to Hermes. The overhead of relocation entries, however, begin to be a limiting factor in contributing to the overall delta size. 3) Overall, R2 reduces the delta sizes further by about 20 percent compared to Hermes. Note that reductions in the transferred code size will result in a similar reduction in the code dissemination time and packet transmissions, which are key metrics for reprogramming effectiveness.

4.4 Reconstruction Overhead

We also evaluate the overhead of patching the delta to generate the new program code. Fig. 10 shows the patching overhead. We can see two facts from the figure. First, R2's overhead is smaller than Hermes due to smaller delta size. Second, the patching overhead is relatively large. However, considering that sending a single bit of data consumes about the same energy as executing 1,000 instructions [25], this overhead is still acceptable.

4.5 Reprogramming Time and Energy

We perform testbed experiments to evaluate the reprogramming time and energy of different approaches in a testbed consisting of 24 TelosB nodes in a 4×6 grid. The internode spacing is about 35 cm, and the transmission power level is configured to 1 to simulate the multihop behavior.

Fig. 11 shows the dissemination time of four approaches including nonincremental approach, native incremental approach, Hermes, and R2. We can see that R2

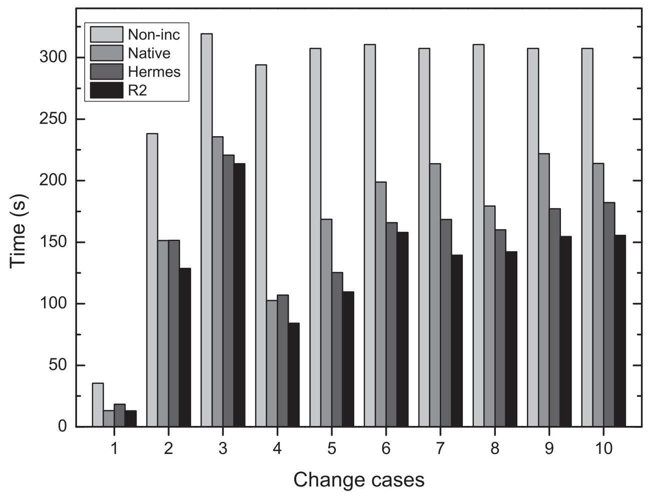


Fig. 11. Dissemination time (s).

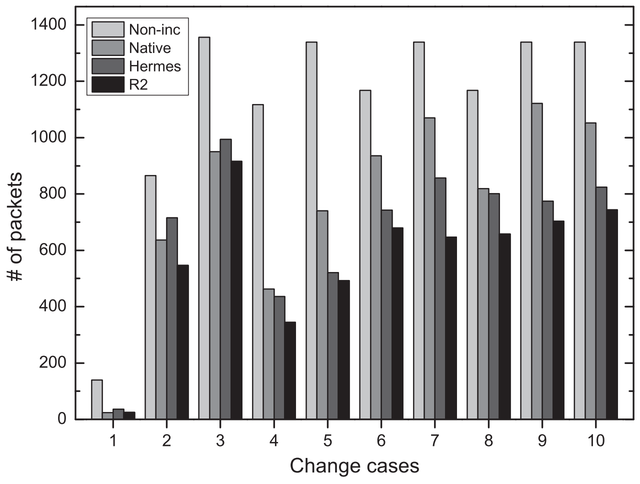


Fig. 12. The number of data transmissions.

reduces the dissemination time by about 20 percent compared to Hermes.

The number of packet transmissions is an important indicator of energy consumption. Similar to [8], we take the number of transmissions as a metric for energy consumption. Fig. 12 shows the number of data packet transmissions of four approaches. We can see that R2 also results in the smallest transmission overhead. R2 reduces the transmission cost by about 20 percent compared to Hermes.

4.6 Summary

We can see that R2 reduces the delta size by approximately 65 percent compared to nonincremental reprogramming and approximately 20 percent compared to Hermes—the latest work on incremental reprogramming.

R2 reduces the delta size without introducing additional complexity compared to Hermes. The delta generation process is performed at the PC side that can accommodate complicated computations. Both Hermes and R2 require binary modifications at the mote side. The binary modification process of R2 is more lightweight than Hermes.

R2 has additional benefits compared to Hermes. 1) R2 unifies the approaches for handling function shifts and data shifts. It also nicely handles the shift of interrupt handlers. 2) R2 does not cause memory segmentation and makes efficient use of memory. R2's symbol table does not occupy program flash space. 3) R2 does not require source-level modification as Hermes. 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. R2 can nicely be integrated to this compilation model.

5 Related Work

This section discusses related work most relevant to our work. We classify existing works into three broad categories: network dissemination protocols, algorithmic optimization techniques, and system optimization techniques.

5.1 Network Dissemination Protocols

Deluge [11] is probably one of the most popular reprogramming protocols used for reliable code updates in wireless ad hoc and sensor networks. It uses a three-way handshake and NACK-based protocol for reliability, and employs segmentation (into pages) and pipelining for spatial multiplexing. It achieves one ninth the maximum transmission rate of the radio supported under TinyOS.

MNP [26] provides a detailed sender selection algorithm to choose a local source of the code that can satisfy the maximum number of nodes. CORD [27] is a more recent work, aiming at minimizing energy consumption. It employs a two-phase approach in which the object is delivered to a subset of nodes in the network that form a connected dominating set in the first phase, and to the remaining nodes in the second phase. ECD [28] improves the code dissemination process by exploiting link quality information.

Several coding-based reprogramming protocols specifically designed for sensor networks are proposed to address the deficiency of Deluge in sparse and lossy networks, such as Rateless Deluge [29], SYNAPSE [30], and AdapCode [31]. They all use network coding to encode a packet before transmission. Upon receiving an expected number of encoded packets, the receiving node uses Gaussian elimination to decode the packets. The difference is that Rateless Deluge [29] uses Random Linear Codes; SYNAPSE [30] uses Fountain Codes; AdapCode [31] also uses linear codes, but the coding scheme is adaptively changed according to the link quality.

5.2 Algorithmic Optimizations

Sadler and Martonosi [32] investigate the design issues involved in adapting compression algorithms specifically geared for sensor nodes. Tsiftes et al. [33] implement gzip for sensor nodes for reducing the size of executable modules. Jeong and Culler [13] modify the Rsync algorithm [21] for efficient incremental reprogramming for sensor nodes. Such a block-based algorithm, suitable for handling large files in computers, is not appropriate for energy-efficient reprogramming for microembedded systems. The RMTD algorithm proposed in [14] is an optimal differencing algorithm with $O(n^3)$ time complexity and $O(n^2)$ space complexity. However, it cannot perform content-aware comparisons, resulting a large delta size.

5.3 Systematic Optimizations

Stream [12] reduces the transferred code size by preinstalling the reprogramming protocol on the external flash as another application image (i.e., the reprogramming image). During the reprogramming process, each node first reboots to the reprogramming image for retrieving the new application code. When the new application code is received, each node reboots again to the new application. Stream is also insufficient for complex applications that usually include a large number of kernel components. Elon [34] addresses this issue by introducing the concept of replaceable component. In the network reprogramming process, only the replaceable component needs to be disseminated, 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.

There are other similarity improvement approaches to enable efficient incremental reprogramming. Li et al. [25] propose a update-conscious register allocation scheme to improve the program similarity. The work is complementary to ours. The works of [6], [7], [8] use specific techniques to mitigate the effects of function shifts and data shifts. As summarized in Section 2, these works have several limitations that need to be addressed.

6 Conclusion

Network reprogramming is of great importance for managing large-scale networked embedded systems, in which incremental reprogramming can greatly reduce the transmission overhead. In this paper, we present R2, an incremental reprogramming system for networked embedded systems.

R2 achieves a higher degree of similarity than existing approaches by mitigating effects of both function shifts and data shifts. R2 adopts a content-aware differencing algorithm to generate small delta files for efficient dissemination. Besides, it makes efficient use of memory and does not degrade program quality. We implement R2 based on TinyOS 2.1 and demonstrate its advantages through detailed analysis of TinyOS examples. We also present case studies on the software programs of a large-scale sensor system—GreenOrbs. Results show that R2 reduces the dissemination cost by approximately 65 percent compared to state-of-the-art network reprogramming approach—Deluge and reduces the dissemination cost by approximately 20 percent compared to Zephyr and Hermes—the latest works on incremental reprogramming.

As future work, we would like to further improve the generality of R2 by porting it to other platforms and other microembedded OSes.

Acknowledgments

This work is supported by the National Science Foundation of China Grants No. 61202402 and 61070155, the China 973 Program under Grant 2011CB302705, the National Key Technology R&D Program (2012BAI34B01), the China 973 Program under Grant 2011CB302705, the Fundamental Research Funds for the Central Universities (2012QNA5007), the Research Fund for the Doctoral Program of Higher Education of China (20120101120179), NSFC Distinguished Young Scholars Program under Grant No. 61125202, and the Program for New Century Excellent Talents in University (NCET-09-0685).

References

[1] I.F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless Sensor Networks: A Survey," *Computer Networks,* vol. 38, pp. 393-422, 2002.

[2] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh, "Fidelity and Yield in a Volcano Monitoring Sensor Networks," *Proc. Seventh Symp. Operating Systems Design and Implementation,* 2006.

[3] Q. Wang, Y. Zhu, and L. Cheng, "Reprogramming Wireless Sensor Networks: Challenges and Approaches," *IEEE Network Magazine,* vol. 20, no. 3, pp. 48-55, May/June 2006.

[4] *GreenOrbs,* http://www.greenorbs.org, 2013.

[5] L. Mo, Y. He, Y. Liu, J. Zhao, S. Tang, X.-Y. Li, and G. Dai, "Canopy Closure Estimates with GreenOrbs: Sustainable Sensing in the Forest," *Proc. ACM Conf. Embedded Networked Sensor Systems,* 2009.

[6] J. Koshy and R. Pandey, "Remote Incremental Linking for Energy-Efficient Reprogramming of Sensor Networks," *Proc. Second European Workshop Wireless Sensor Networks,* 2005.

[7] R.K. Panta, S. Bagchi, and S.P. Midkiff, "Zephyr: Efficient Incremental Reprogramming of Sensor Nodes Using Function Call Indirections and Difference Computation," *Proc. USENIX Ann. Technical Conf.,* 2009.

[8] R.K. Panta and S. Bagchi, "Hermes: Fast and Energy Efficient Incremental Code Updates for Wireless Sensor Networks," *Proc. IEEE INFOCOM,* 2009.

[9] J.R. Levine, *Linkers and Loaders.* Morgan Kaufmann, 2000.

[10] *TinyOS,* http://www.tinyos.net, 2013.

[11] J.W. Hui and D. Culler, "The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale," *Proc. ACM Conf. Embedded Networked Sensor Systems,* 2004.

[12] R.K. Panta, I. Khalil, and S. Bagchi, "Stream: Low Overhead Wireless Reprogramming for Sensor Networks," *Proc. IEEE INFOCOM,* 2007.

[13] J. Jeong and D. Culler, "Incremental Network Programming for Wireless Sensors," *Proc. First Ann. IEEE Comm. Soc. Conf. Sensor and Ad Hoc Comm. Networks,* 2004.

[14] J. Hu, C.J. Xue, and Y. He, "Reprogramming with Minimal Transferred Data on Wireless Sensor Network," *Proc. IEEE Sixth Int'l Conf. Mobile Adhoc and Sensor Systems,* 2009.

[15] M. Maróti, B. Kusy, G. Simon, and Á. Lédeczi, "The Flooding Time Synchronization Protocol," *Proc. ACM Conf. Embedded Networked Sensor Systems,* 2004.

[16] T. He, P. Vicaire, T. Yan, Q. Cao, G. Zhou, L. Gu, L. Luo, R. Stoleru, J.A. Stankovic, and T. Abdelzaher, "Achieving Long-Term Surveillance in VigilNet," *Proc. IEEE INFOCOM,* 2006.

[17] H. Pucha, D.G. Andersen, and M. Kaminsky, "Exploiting Similarity for Multi-Source Downloads Using File Handprints," *Proc. USENIX Conf. Networked Systems Design and Implementation,* 2007.

[18] F. Dabek, M.F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-Area Cooperative Storage with CFS," *Proc. ACM Symp. Operating Systems Principles,* 2001.

[19] F. Douglis and A. Iyengar, "Application-Specific Delta-Encoding via Resemblance Detection," *Proc. USENIX Ann. Technical Conf.,* 2001.

[20] K. Tangwongsan, H. Pucha, D.G. Andersen, and M. Kaminsky, "Efficient Similarity Estimation for Systems Exploiting Data Redundancy," *Proc. IEEE INFOCOM,* 2010.

[21] *Rsync,* http://samba.anu.edu.au/rsync/, 2013.

[22] P. Levis, N. Patel, D. Culler, and S. Shenker, "Trickle: A Self-Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks," *Proc. Conf. Symp. Networked Systems Design and Implementation,* 2004.

[23] G. Tolle and D. Culler, "Design of an Application-Cooperative Management System for Wireless Sensor Networks," *Proc. European Workshop Wireless Sensor Networks,* 2005.

[24] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis, "Collection Tree Protocol," *Proc. ACM Conf. Embedded Networked Sensor Systems,* 2009.

[25] W. Li, Y. Zhang, J. Yang, and J. Zheng, "UCC: Update-Conscious Compilation for Energy Efficiency in Wireless Sensor Networks," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* 2007.

[26] S.S. Kulkarni and L. Wang, "MNP: Multihop Network Reprogramming Service for Sensor Networks," *Proc. 25th IEEE Int'l Conf. Distributed Computing Systems,* 2005.

[27] L. Huang and S. Setia, "CORD: Energy-Efficient Reliable Bulk Data Dissemination in Sensor Networks," *Proc. IEEE INFOCOM,* 2008.

[28] W. Dong, Y. Liu, C. Wang, X. Liu, C. Chen, and J. Bu, "Link Quality Aware Code Dissemination in Wireless Sensor Networks," *Proc. IEEE Int'l Conf. Network Protocols,* 2011.

[29] A. Hagedorn, D. Starobinski, and A. Trachtenberg, "Rateless Deluge: Over-the-Air Programming of Wireless Sensor Networks Using Random Linear Codes," *Proc. IEEE Int'l Conf. Information Processing in Sensor Networks,* 2008.

[30] M. Rossi, N. Bui, G. Zanca, L. Stabellini, R. Crepaldi, and M. Zorzi, "SYNAPSE++: Code Dissemination in Wireless Sensor Networks Using Fountain Codes," *IEEE Trans. Mobile Computing,* vol. 9, no. 12, pp. 1749-1765, Dec. 2010.

[31] I.-H. Hou, Y.-E. Tsai, T.F. Abdelzaher, and I. Gupta, "AdapCode: Adaptive Network Coding for Code Updates in Wireless Sensor Networks," *Proc. IEEE INFOCOM,* 2008.

[32] C.M. Sadler and M. Martonosi, "Data Compression Algorithms for Energy-Constrained Devices in Delay Tolerant Networks," *Proc. ACM Conf. Embedded Networked Sensor Systems,* 2006.

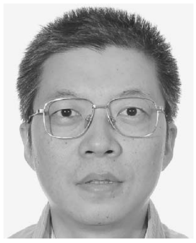
[33] N. Tsiftes, A. Dunkels, and T. Voigt, "Efficient Sensor Network Reprogramming through Compression of Executable Modules," *Proc. Fifth Ann. IEEE Comm. Soc. Conf. Sensor, Mesh and Ad Hoc Comm. Networks,* 2008.

[34] W. Dong, Y. Liu, X. Wu, L. Gu, and C. Chen, "Elon: Enabling Efficient and Long-Term Reprogramming for Wireless Sensor Networks," *Proc. ACM SIGMETRICS Int'l Conf. Measurement and Modeling of Computer Systems,* 2010.

Wei Dong received the BS and PhD degrees in computer science from Zhejiang University in 2005 and 2010, respectively. He was a postdoc fellow in the Department of Computer Science and Engineering, Hong Kong University of Science and Technology in the year 2011. He is currently an assistant professor at the College of Computer Science, Zhejiang University. His research interests include networked embedded systems and wireless sensor networks. He is a member of the IEEE.

Yunhao Liu received the BS degree from the Automation Department, Tsinghua University, China, in 1995, and the MA degree from Beijing Foreign Studies University, China, in 1997, and the MS and PhD degrees in Computer Science and Engineering from Michigan State University in 2003 and 2004, respectively. He is currently a professor with the School of Software, Tsinghua University. His research interests include distributed systems and P2P, RFID and wireless sensor networks, privacy and security. He is a senior member of IEEE, and a distinguished speaker of the ACM.

Chun Chen received the bachelor of mathematics degree from Xiamen University, China, in 1981, and the MS and PhD degrees in computer science from Zhejiang University, China, in 1984 and 1990, respectively. He is a professor at the College of Computer Science, and the Director of the Institute of Computer Software, Zhejiang University. His research interests include embedded system, image processing, computer vision, and CAD/CAM. He is a member of the IEEE.



Jiajun Bu received the BS and PhD degrees in computer science from Zhejiang University, China, in 1995 and 2000, respectively. He is a professor at the College of Computer Science and the deputy dean in the Department of Digital Media and Network Technology, Zhejiang University. His research interests include embedded system, mobile multimedia, and data mining. He is a member of the IEEE and the ACM.

Chao Huang received the BS degree in computer science from the Dalian University of Technology in 2010, and the MS degree in computer science from Zhejiang University in 2013. He is currently a software engineer of Alibaba Group. His research interests include sensor networks and mobile computing.

Zhiwei Zhao received the BS degree at the College of Electronic and Information, Xi'an Jiaotong University in 2010. He is currently working toward the PhD degree at the College of Computer Science, Zhejiang University. His research interests mainly focus on wireless sensor networks. He is a student member of the IEEE.