This paper was presented as part of the Mini-Conference at IEEE INFOCOM 2011

$\mathcal{R}^2$: Incremental $\mathcal{R}$eprogramming using $\mathcal{R}$elocatable Code in Networked Embedded Systems

Wei Dong[†‡], Yunhao Liu[‡], Chun Chen[†], Jiajun Bu[†], and Chao Huang[†]
[†]Zhejiang Key Laboratory of Service Robot,
College of Computer Science, Zhejiang University
[‡]Department of Computer Science, HKUST
{dongw, chenc, bjj}@zju.edu.cn, liu@cse.ust.hk

Abstract—We present $\mathcal{R}^2$, an incremental $\mathcal{R}$eprogramming approach using $\mathcal{R}$elocatable code, to improve program similarity for efficient incremental reprogramming in networked embedded systems. $\mathcal{R}^2$ achieves a higher degree of similarity than existing approaches by mitigating the effects of both function shifts and data shifts. $\mathcal{R}^2$ makes efficient use of memory and does not degrade program quality. It adopts an optimized differencing algorithm to generate small delta files for efficient dissemination. We demonstrate $\mathcal{R}^2$'s advantages through detailed analysis of TinyOS examples. We also present case studies on the software programs of a large-scale and long-term sensor system—GreenOrbs. Results show that $\mathcal{R}^2$ reduces the dissemination cost by approximately 65% compared to Deluge—state-of-the-art network reprogramming approach, and reduces the dissemination cost by approximately 20% compared to Zephyr and Hermes—the latest works on incremental reprogramming.

I. 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 [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 module parameters, etc. 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 re-deploying all nodes in the field.

In the network reprogramming process, the new program code needs to be disseminated to all nodes in the network. Reducing the transferred code size is highly desirable for shortening the reprogramming time, reducing the transmission overhead, and reducing the energy consumption.

Incremental reprogramming is a technique to reduce the code dissemination overhead. In this approach, only the delta between the new code and the old code is generated and disseminated. In this work, we propose a novel incremental reprogramming approach to effectively reduce the delta size by exploiting the similarity between two program versions. The key idea to improve the 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 change, resulting in decreased similarity between two program versions. Similar problems exist for global data variables.

Koshy and Pandey [6] 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 [7] mitigates the effects of function shifts by using a function jump 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). Moreover, it decreases the execution efficiency because of indirections. Although the program can be optimized to use direct function calls at load time, as proposed in Hermes [8], the loader implementation is complicated as it has to handle variable length instructions in a platform-specific manner (e.g., the instructions for TelosB nodes are of variable length), making portability to different platforms difficult.

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 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 impractical for memory-intensive and long-term applications. For the Harvard-based MicaZ nodes, it 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

978-1-4244-9921-2/11/$26.00 ©2011 IEEE

`.data` variables up to the slop region boundary.

We propose $\mathcal{R}^2$, a unified approach to mitigate both effects of *function shifts* and *data shifts* by using *relocatable code*. This approach obtains a higher degree of similarity by keeping all references in the instructions the same in both program versions. Moreover, it makes efficient use of the memory and does not degrade the program quality.

We examine the performance of $\mathcal{R}^2$ through analysis of TinyOS examples as well as results obtained from software programs for our 330-node sensor system—GreenOrbs. Results show that (i) $\mathcal{R}^2$ improves the program similarity compared to existing approaches. (ii) $\mathcal{R}^2$ preserves program quality in terms of memory efficiency and execution efficiency. (iii) $\mathcal{R}^2$ reduces the delta size by approximately 20% compared to Hermes [8] in our case studies of the GreenOrbs programs.

The rest of this paper is organized as follows. Section II presents the design of $\mathcal{R}^2$. Section III shows the evaluation results. Section IV summarizes the related work. Finally, we conclude this study in Section V.

II. Design

Our overall design goal is to develop a useful incremental reprogramming approach—$\mathcal{R}^2$, that achieves minimum transmission overhead. Our approach should address various limitations of existing approaches and have the following features.

- High degree of similarity. $\mathcal{R}^2$ should handle a diversity of reference instructions, e.g., `call`, `mov`, `br`, etc.
- Unified approach. $\mathcal{R}^2$ should unify the approaches for handling function shifts and data shifts.
- High program quality. $\mathcal{R}^2$ should preserve the high program quality in terms of run-time execution efficiency and memory efficiency.
- Generality. $\mathcal{R}^2$ should be portable to multiple platforms and compilation models.
- Small delta size. $\mathcal{R}^2$ 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 $\mathcal{R}^2$ should meet the resource constraints of micro embedded systems.

The main idea of applying relocatable code is to make all the 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. In order to make these reference instructions behave correctly at run time, we must generate additional metadata, i.e., relocation entries, for a loader software residing on the nodes to perform load-time modifications. In general, a relocation entry contains the following information (i) at which address to apply the modification, and (ii) the correct target address for the reference. This is a common technique for traditional dynamic linking and loading [9] because the actual loading addresses cannot be determined at link time. 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 target addresses can be different between two program versions.

The advantages of utilizing relocatable code are as follows.

- It is directly supported by the compiler and linker. For example, for compiling TinyOS applications, a simple linker option of "`-Wl,-q`" can generate relocatable entries in the final ELF file.
- 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 run-time execution efficiency and memory efficiency: (i) it does incur indirection costs; (ii) no slop regions are needed and data can be compacted when some variables are deleted in the new version.

We modify the code generation mechanism of TinyOS/nesC to use 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 entries into a binary file for delta generation.

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 address to apply the relocation. `r_info`, together with `r_addend`, tells information on how to apply the relocation (i.e., what is the target address). For example, if the reference instruction refers to a symbol, `r_info` can be used to locate the symbol for its 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 micro embedded systems, it is important to compress the relocation entry. Specifically, relocation entry in $\mathcal{R}^2$ is of the following type,

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

where `r_offset` tells the address 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 4 bytes.

It is worth noting that the difference of this design with the jump table approach of Zephyr [7]. In Zephyr, each jump table slot consumes 6 bytes (a `call` instruction consumes 4 bytes and a `ret` instruction consumes 2 bytes). However, the number of jump table slots is less than the number of relocation entries since it handles less types of reference instructions. The tradeoff is that it results in a lower degree of similarity which increases the delta size. In our design, the size of the relocation entries can be optimized by exploiting some special patterns.

We now present our differencing algorithm for generating the delta. The delta file is used to reconstruct the new code by inserting new bytes or copying bytes from the old code. For the relocatable code, we adopt an optimal differencing algorithm similar to the RMTD algorithm [10]. This algorithm compares the old program and the new program at the byte level, generating the smallest delta size. For the relocation entries, we adopt content-aware comparison to further reduce the delta size.

We find that it is common that,

- `r_offset`s shift for a constant in a consecutive of entries while `r_addr`s keep the same as the old version.
- `r_addr`s shift for a constant in a consecutive of entries while `r_offset`s keep the same as the old version.
- both `r_addr`s and `r_offset`s shift for some constants.

In the above circumstances, the original RMTD algorithm cannot copy a consecutive of entries. We devise additional delta-commands in order to copy a consecutive of entries with a constant shift.

III. Evaluation

This section evaluates the effectiveness of our design. Section III-A introduces our evaluation methodology. Section III-B analyzes a specific TinyOS application to show the advantages of $\mathcal{R}^2$ in improving the program similarity. Section III-C presents case studies in the development of GreenOrbs to illustrate the end-to-end benefits of $\mathcal{R}^2$ in improving reprogramming efficiency.

A. Methodology

We evaluate $\mathcal{R}^2$ based on TinyOS 2.1 for TelosB nodes. The TelosB node uses the msp430f1611 microcontroller with variable length instructions. It has 10KB RAM for storing global data (also the program stack) and 48KB for storing program code. It has an additional 1MB external flash for storing persistent data. Towards implementing $\mathcal{R}^2$, we write codes for (i) the relocatable code generation methods on the PC side (in Perl), (ii) the optimized differencing algorithm on the PC side (in C).

We first analyze small change cases to the basic `Blink` application in TinyOS 2.1 to show the effectiveness of $\mathcal{R}^2$'s similarity improvement approach. The `Blink` application blinks three leds with three timers of firing intervals 250ms, 500ms, 1000ms, respectively.

We then study two real-world change cases in the development of GreenOrbs to demonstrate the effectiveness of $\mathcal{R}^2$'s approach 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 [11] for collecting multiple types of sensor data, e.g., light, temperature, humidity, to a collection sink. To increase the flexibility, GreenOrbs includes the Drip component [12] for disseminating key system parameters, e.g., duty cycle ratio, transmission power settings. To achieve energy efficiency, GreenOrbs also includes the FTSP component [13] for enabling synchronous low duty cycling.

B. Examples

In this section, we look at the `Blink` example in TinyOS 2.1. We consider a change case from `Blink` to `CntToLeds`.

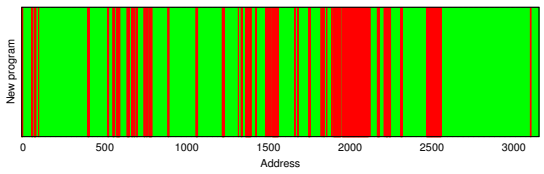
The bytes in the new program code that cannot be found in a common segment in the old program code must be encoded in the `ADD` command in the delta file. Thus, added bytes measure the degree of program similarity. We compare similarity achieved by three approaches: (i) native: optimal diff without any similarity improvement methods, (ii) Hermes: optimal diff with Hermes optimization, (iii) $\mathcal{R}^2$: optimal diff with $\mathcal{R}^2$ optimization. We have found that $\mathcal{R}^2$ achieves the highest degree of similarity: it requires 82 added bytes while the native approach requires 478 bytes and Hermes requires 134 bytes. $\mathcal{R}^2$ 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, e.g., the branch instruction `br`, the `mov` instruction, interrupt services routines (from address `0xffe0` to address `0xffff`).

$\mathcal{R}^2$ achieves better similarity when data is added to a program. In Hermes, when some data expand the pre-defined 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. Figure 1 shows the added bytes at different locations in the new program. We can see that while $\mathcal{R}^2$ only slightly decreases the similarity (the number of added bytes increases from 82 bytes to 93 bytes), Hermes decreases the similarity significantly (the number of added bytes increases from 134 bytes to 595 bytes). Hence $\mathcal{R}^2$ is more general in unifying the similarity improvement techniques for handling both functions shifts and data shifts, achieving a higher degree of similarity than existing approaches.

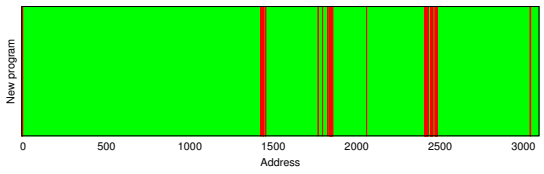
C. Case Studies

We examine the delta size generated by $\mathcal{R}^2$ compared to Zephyr/Hermes [7], [8] via two change cases in our development of GreenOrbs.

- Case A: in this case, we change the GreenOrbs program from SVN version 15 to SVN version 16. Version 15



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



(b) $\mathcal{R}^2$ (93 bytes need to be added)

Fig. 1: Similarity comparison of two approaches

integrates with the CTP protocol [11], collecting temperature, humidity, light, and voltage. Version 16 adds the functionality of `printf` to enable effective debugging.

- Case B: in this case, we change the GreenOrbs program from SVN version 109 to SVN version 138. Version 109 integrates with CTP [11], Drip [12], and FTSP [13], collecting temperature, humidity, light, and voltage at a duty cycle of approximately 5%. Version 138 has three kinds of updates. (i) update the FTSP protocol to filtering the error readings in the CC2420 driver. (ii) log key system events onto the external flash. (iii) in each cycle, transfer another (diagnostic) packet to the sink, with information such as CTP parent, accumulated retransmission count, accumulated sent/received number of packets, etc.

To examine how $\mathcal{R}^2$ improves the efficiency of state-of-arts reprogramming protocols, we evaluate (i) non-incremental reprogramming by disseminating the new program code. (ii) the deltas generated without optimizing the similarity. (iii) the deltas generated by Hermes. (iv) the deltas generated by $\mathcal{R}^2$.

Figure 2 shows the comparison results. We can see that (i) incremental reprogramming is effective in reducing the transferred code sizes, resulting in 37.3%, 47.6% reductions in the transferred bytes (for Case A, and Case B, respectively). (ii) Hermes' similarity improvement approach can further reduce the delta sizes by 28.3%, and 15.6% respectively, compared to using the optimal diff algorithm only. (iii) $\mathcal{R}^2$ significantly improves the similarity compared to the examples we investigated in the previous subsection, while the overhead of relocation entries begin to be a limiting factor in contributing to the overall delta size. (iv) Overall, $\mathcal{R}^2$ reduces the delta sizes further by 19.4%, and 19.5%, respectively, 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.

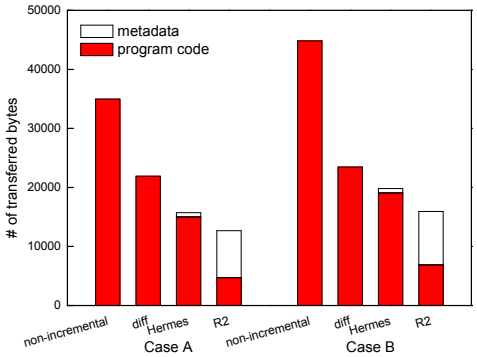


Fig. 2: Delta size comparison of four reprogramming approaches for change Case A and Case B

IV. Related Work

This section discusses related work most relevant to our work.

Network Dissemination Protocols. Deluge [14] 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 [15] provides a detailed sender selection algorithm to choose a local source of the code which can satisfy the maximum number of nodes. CORD [16] 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.

Recently, 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 [17], SYNAPSE [18], AdapCode [19] and ReXOR [20]. They all use network coding to encode a packet before transmission. Upon receiving an expected number of encoded packets, the receiving node decodes the packets.

Algorithmic Optimizations. Sadler and Martonosi [21] investigate the design issues involved in adapting compression algorithms specifically geared for sensor nodes. Tsiftes *et al.* [22] implement GZIP for sensor nodes for reducing the size of executable modules. Jeong and Culler [23] modify the Rsync algorithm [24] 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 micro embedded systems. The RMTD algorithm proposed in [10] is an optimal differencing algorithm with $O(n^2)$ time complexity. Ajtai *et al.* [25] introduce a greedy optimal differencing algorithm. However, it is no longer optimal under our realistic cost measure, and

must be re-devised for achieving the optimal results.

Systematic Optimizations. Stream [26] reduces the transferred code size by pre-installing 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 which usually include a large number of kernel components. Elon [27] 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.* [28] propose a update-conscious register allocation scheme to improve the program similarity. The work is complementary to ours. The works of [6]–[8] use specific techniques to mitigate the effects of function shifts and data shifts. As described in Section I, these works have several limitations that should be addressed properly.

V. Conclusion

Network reprogramming is of great importance for managing large-scale networked embedded systems, in which incremental reprogramming can greatly reduce the transmission overhead. Existing incremental reprogramming approaches suffer from major limitations of (i) limited similarity, and (ii) degraded program quality.

To address these limitations, we present $\mathcal{R}^2$, an incremental reprogramming approach for networked embedded systems. $\mathcal{R}^2$ achieves a higher degree of similarity than existing approaches by mitigating the effects of both function shifts and data shifts. $\mathcal{R}^2$ makes efficient use of memory and does not degrade the program quality. It adopts an optimized differencing algorithm to generate small delta files for efficient dissemination.

Future work leads to two directions. First, we will improve the visibility of reliability of $\mathcal{R}^2$ for managing large-scale ad hoc networks like our 330 nodes GreenOrbs in the forest. Second, we will further improve the generality of $\mathcal{R}^2$ by porting it to other platforms and other micro embedded OSes.

Acknowledgments

We thank J. Hu for providing us the RMTD code. This work is supported by the National Basic Research Program of China (973 Program) under grant No. 2006CB303000, the National Science Foundation of China (Grant No. 61070155), 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," in *Proc. of USENIX OSDI*, 2006.
[3] Q. Wang, Y. Zhu, and L. Cheng, "Reprogramming Wireless Sensor Networks: Challenges and Approaches," *IEEE Network Magazine*, vol. 20(3), pp. 48–55, 2006.
[4] GreenOrbs: http://www.greenorbs.org.
[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," in *Proc. of ACM SenSys*, 2009.
[6] J. Koshy and R. Pandey, "Remote Incremental Linking for Energy-Efficient Reprogramming of Sensor Networks," in *Proc. of EWSN*, 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," in *Proc. of USENIX Annual Technical Conference*, 2009.
[8] R. K. Panta and S. Bagchi, "Hermes: Fast and Energy Efficient Incremental Code Updates for Wireless Sensor Networks," in *Proc. of IEEE INFOCOM*, 2009.
[9] J. R. Levine, *Linkers and Loaders*. Morgan Kaufmann, 2000.
[10] J. Hu, C. J. Xue, and Y. He, "Reprogramming with Minimal Transferred Data on Wireless Sensor Network," in *Proc. of IEEE MASS*, 2009.
[11] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis, "Collection Tree Protocol," in *Proc. of ACM SenSys*, 2009.
[12] G. Tolle and D. Culler, "Design of an Application-Cooperative Management System for Wireless Sensor Networks," in *Proc. of EWSN*, 2005.
[13] M. Maróti, B. Kusy, G. Simon, and Á. Lédeczi, "The Flooding Time Synchronization Protocol," in *Proc. of ACM SenSys*, 2004.
[14] J. W. Hui and D. Culler, "The dynamic behavior of a data dissemination protocol for network programming at scale," in *Proc. of ACM SenSys*, 2004.
[15] S. S. Kulkarni and L. Wang, "MNP: Multihop Network Reprogramming Service for Sensor Networks," in *Proc. of IEEE ICDCS*, 2005.
[16] L. Huang and S. Setia, "CORD: Energy-efficient Reliable Bulk Data Dissemination in Sensor Networks," in *Proc. of IEEE INFOCOM*, 2008.
[17] A. Hagedorn, D. Starobinski, and A. Trachtenberg, "Rateless Deluge: Over-the-Air Programming of Wireless Sensor Networks using Random Linear Codes," in *Proc. of ACM/IEEE IPSN*, 2008.
[18] M. Rossi, G. Zanca, L. Stabellini, R. Crepaldi, A. F. H. III, and M. Zorzi, "SYNAPSE: A Network Reprogramming Protocol for Wireless Sensor Networks using Fountain Codes," in *Proc. of IEEE SECON*, 2008.
[19] I.-H. Hou, Y.-E. Tsai, T. F. Abdelzaher, and I. Gupta, "AdapCode: Adaptive Network Coding for Code Updates in Wireless Sensor Networks," in *Proc. of IEEE INFOCOM*, 2008.
[20] W. Dong, C. Chen, X. Liu, J. Bu, and Y. Gao, "A Lightweight and Density-aware Reprogramming Protocol for Wireless Sensor Networks," *IEEE Transactions on Mobile Computing*, (to appear).
[21] C. M. Sadler and M. Martonosi, "Data Compression Algorithms for Energy-Constrained Devices in Delay Tolerant Networks," in *Proc. of ACM SenSys*, 2006.
[22] N. Tsiftes, A. Dunkels, and T. Voigt, "Efficient Sensor Network Reprogramming through Compression of Executable Modules," in *Proceedings of IEEE SECON*, 2008.
[23] J. Jeong and D. Culler, "Incremental Network Programming for Wireless Sensors," in *Proc. of IEEE SECON*, 2004.
[24] Rsync: http://samba.anu.edu.au/rsync/.
[25] M. Ajtai, R. Burns, R. Fagin, D. D. E. Long, and L. Stockmeyer, "Compactly encoding unstructured inputs with differential compression," *Journal of the ACM*, vol. 49, no. 3, pp. 318–367, 2002.
[26] R. K. Panta, I. Khalil, and S. Bagchi, "Stream: Low Overhead Wireless Reprogramming for Sensor Networks," in *Proc. of IEEE INFOCOM*, 2007.
[27] W. Dong, Y. Liu, X. Wu, L. Gu, and C. Chen, "Elon: Enabling Efficient and Long-Term Reprogramming for Wireless Sensor Networks," in *Proc. of ACM SIGMETRICS*, 2010.
[28] W. Li, Y. Zhang, J. Yang, and J. Zheng, "UCC: Update-Conscious Compilation for Energy Efficiency in Wireless Sensor Networks," in *Proc. of ACM PLDI*, 2007.