Towards Update-Conscious Compilation for Energy-Efficient Code Dissemination in WSNs

WEIJIA LI, YOUTAO ZHANG, JUN YANG, and JIANG ZHENG

University of Pittsburgh

Postdeployment code dissemination in wireless sensor networks (WSN) is challenging, as the code has to be transmitted via energy-expensive wireless communication. In this article, we propose novel update-conscious compilation (UCC) techniques to achieve energy efficiency. By integrating the compilation decisions in generating the old binary, an update-conscious compiler strives to match the old decisions, which improves the binary code similarity, reduces the amount of transmitted data to remote sensors, and thus, consumes less energy. In this article, we develop update-conscious register allocation and data layout algorithms. Our experimental results show great improvements over the traditional, update-oblivious approaches.

Categories and Subject Descriptors: D.3.4 [Programming Languages]: Processors—Compilers; C.2.3 [Network Operations]: Network Management—Reprogramming

General Terms: Design, Languages, Performance

Additional Key Words and Phrases: Register allocation, sensor networks, code dissemination

ACM Reference Format:

Li, W., Zhang, Y., Zheng, J., and Yang, J. 2009. Towards update-conscious compilation for energy-efficient code dissemination in WSNs. *ACM Trans. Architect. Code Optim.* 6, 4, Article 14 (October 2009), 33 pages. DOI = 10.1145/1596510.1596512 <http://doi.acm.org/10.1145/1596510.1596512>

1. INTRODUCTION

The wireless sensor network (WSN) [Callaway 2003; Juang et al. 2002; Kahn et al. 2000] has recently emerged as a promising computing platform for many nontraditional applications, such as wildfire monitoring in forests, and intelligence surveillance in a battle field. A WSN usually consists of hundreds or thousands of low-cost, battery-powered sensor nodes that are preloaded with

This work is partially supported by NSF grants CCF-0541456 and CAREER CCF-0641177.

Authors' address: W. Li and J. Zheng, Computer Science Department, University of Pittsburgh, Pittsburgh PA 15260; Y. Zhang and J. Yang, Electrical and Computer Engineering Department, University of Pittsburgh, Pittsburgh PA 15261; email: {weijiali,zhangyt,jzheng}@cs.pitt.edu, juy9@pitt.edu

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2009 ACM 1544-3566/2009/10-ART14 \$10.00

DOI 10.1145/1596510.1596512 <http://doi.acm.org/10.1145/1596510.1596512>

ACM Transactions on Architecture and Code Optimization, Vol. 6, No. 4, Article 14, Pub. date: October 2009.

application code and data and then deployed into the field to track events of interest. Sensing results are constructed into data packets, and routed back to a sink node, which is typically more powerful, user accessible, and has fewer energy constraints. In contrast, the sensor nodes are usually left unattended after deployment, so they are extremely energy and storage constrained. Of all constraints in a WSN, the energy constraint is more predominant and largely determines the lifetime of the network.

Due to the changes of user requirements and environmental conditions, the preloaded program code and data on wireless sensors often need to be updated. For example, a WSN may be deployed in a field where scientists have very limited knowledge of, for example, deep ocean or wild nature. In those environments, people first collect and analyze the field data and then develop more effective sensing functions to process more interesting and important phenomena. Such functions are often missing in the preloaded code. A WSN can also be deployed inside a building to detect water damage, sound propagation, earthquake damage, and the like. Preloading all functions into the sensors is infeasible due to their limited memory sizes. In addition, it may also be infeasible to deploy a new WSN for every new task. Hence, reprogramming sensors on demand is more economical and practical [Levis and Culler 2002].

Since sensor nodes are left unattended (or even become unreachable) after deployment, reprogramming can only be done through wireless communication, which is expensive in terms of energy consumption. For large WSNs where the sink cannot reach every node through broadcasting, updates can only be transmitted hop-by-hop within the WSN, consuming significant energy. Recent studies have shown that sending a single bit of data consumes about the same energy as executing 1,000 instructions [Shnayder et al. 2004; Reijers and Langendoen 2003]. As a result, it is essential to conserve the energy in a WSN during the code and data dissemination, especially when the update happens frequently.

The current code dissemination approaches can be categorized according to what is to be transmitted over the network. The simplest solution, employed by Deluge (the default code distribution scheme in TinyOS [2008]), is to transmit the complete updated binary code to replace the old version on sensors. Another approach, the *diff*-based design, compares the code of successive versions and generates an edit script that summarizes the differences. Only the script is transmitted to the remote sensor where the new code is regenerated from both the old image and the edit script. Since less data is transmitted over the network and the edit script is usually simple and can be easily interpreted by the sensor, the *diff*-based approach significantly improves energy efficiency and has become more popular in WSNs [Reijers and Langendoen 2003; Panta et al. 2007; Jeong and Culler 2004; Marron et al. 2006; Koshy et al. 2005; Dunkels et al. 2006].

Code can also be disseminated at different levels. Some recent work introduced a small virtual machine [Levis and Culler 2002] or a dynamic linker [Dunkels et al. 2006; Koshy et al. 2005] on remote sensors. Instead of binary instructions, the code is represented at a higher level (e.g., virtual machine primitives), which can minimize the code difference in many cases. The trade-off is

that such approaches introduce high-runtime overhead and may consume more energy in the long run.

Though the concept of incremental update was incorporated in the previously described approaches, the code differences are derived from binaries generated using the conventional compiler's code generation methods, with possibly some optimizations. Therefore, a simple change in the source code may result in many changes in the final binary. This has limited the *diff*-based approaches to only small updates, such as fixing a bug [Reijers and Langendoen 2003].

In this article, we propose update-conscious compilation (UCC) techniques that target at improving the code similarity between the binary code and its previous version. Specifically, we attempt to minimize the instruction differences so that the update transmission over a WSN is greatly reduced. However, this may result in a compromise in code execution time, which is a concern, since the new code is executed on remote sensors hereafter. We consider this trade-off and generate the new code in a way that the overall energy consumption is reduced in the long run. After generating the new code, the differences are summarized in a small script, which is then transmitted to the remote sensor. The new code is generated on the remote sensor through interpreting the update script to change the old binary. This concludes the code dissemination process.

The remainder of the article is organized as follows. The overview of UCC is presented in Section 2. We discuss update-conscious register allocation in Section 3, and update-conscious data allocation in Section 4. The code dissemination is discussed in Section 5. The experimental results are presented in Section 6. More related work is discussed in Section 7. Finally, we conclude this article in Section 8.

2. OVERVIEW

The conventional compilation takes the following steps to generate a binary code from the source code, as depicted in Figure 1. First, the compiler converts the source code S into an intermediate representation ir . Next, the compiler optimizes the ir for several iterations, and produces the optimized intermediate representation IR . Finally, the code generation stage uses IR to generate the binary code E by applying data allocation, code placement, register allocation, and the like.

Our proposed UCC is performed at the code generation stage (i.e., from IR to E). This helps to preserve the performance improvements from the optimization passes. In this article, we focus on the register allocation and data allocation techniques. For clarity, we assume that the optimization passes are independent of register allocation and data allocation, and other optimizations will be investigated in our future work.

When S is updated to S' (Figure 1), ir and IR are also updated to ir' and IR' , respectively. Let Δ represent the differences between the IR' and its previous version IR . With Δ , the compiler can analyze and decide how to generate the binary E' such that its difference from E , denoted as U , is small. The decision is made by considering the energy gain and cost from transmitting the code

Fig. 1. The sink-side update-aware compilation.

Fig. 2. The sensor-side code update and execution.

versus executing the code on remote sensors. Finally, when E' is generated, U is produced and summarized in a script, which is then disseminated to the sensors.

The code update on remote sensors is relatively simple, as illustrated in Figure 2. The update script U is interpreted locally to change the old binary E to E' . Changes may include inserting/removing instructions, constant updates, register name replacement, and the like. As a comparison, previous schemes such as code update with virtual machine on sensors [Levis and Culler 2002] or using a dynamic linker [Dunkels et al. 2006] do not store the executables in sensors. Thus, they tend to introduce much more runtime overhead and consume more energy than our binary level update scheme.

2.1 The Power Model and its Application in Compilation

While improving the code similarity is our goal, it is also essential to consider its impact on energy consumption when the code is executed on remote sensors. In general, improved code similarity results in a smaller update script, and thus less transmission energy consumption. In some cases, slightly slower code may still have better energy efficiency. For example, it might be worth the effort to reduce the update script by one word because the new code is not executed very frequently but is very necessary on the sensors. One can also argue for a counterexample. To achieve a good balance between transmission energy and execution energy, we need to first develop a power model for our framework.

We select the Mica2 Mote [Xbow] as our test bed while the techniques are applicable to other types of sensors as well. Mica2 Mote includes a 7.3MHz

Mode	Current	Mode	Current
CPU active	8.0mA	Radio Rx	7 mA
CPU idle	3.2mA	Tx(+10dB)	21.5mA
CPU Standby	216 μ A	EEPROM read	6.2mA
LEDs	2.2mA	EEPROM write	18.4mA

Fig. 3. The power model for Mica2.

CPU, 128KB program flash memory, 512KB measurement flash memory, and 4KB configuration EEPROM. It can transmit data at 38.4Kbps. For Mica2 Motes, transmitting a data bit takes more CPU cycles and more overhead than executing an instruction. In Figure 3, we show the current that the sensor draws at different operational modes [Shnayder et al. 2004]. From these parameters, prior work [Shnayder et al. 2004] showed that for a typical battery capacity of 2700mAh, a Mica2 node that stays active for about 15 minutes per day can last for about one year. In such a sensor network, transmitting more data adds buffering overhead and increases the possibility of signal collision.

Recent studies [Reijers and Langendoen 2003; Barr and Asanović 2003] showed that for such sensors, transmitting a single bit consumes about 1,000 times more energy than executing an ALU instruction.

Next, we collect program execution profiles to estimate how often an updated code will be in use. This will help make good update decisions. For example, assume we need to make a decision whether to add one more instruction in the final binary but save one instruction word in transmission. It is overall energy efficient only if the new instruction is executed less than 16,000 times (16-bit word width $\times$ 1,000).

Let us consider another example that requires the knowledge of the target WSN. Typical sensors need to accomplish two types of tasks: data processing and data transmission. Thus, the corresponding program code can be categorized into two types as well. For large multihop WSNs that have thousands of nodes, a data report may jump 70 or more hops before reaching the sink [Ye et al. 2005]. An interesting event may invoke the data processing code in the originating sensor once but the data transmission code 70 times along the path to the sink. As a result, it is more energy efficient to update data processing code with the highest similarity to its previous version, but update data transmission code with one that consumes the lowest energy (and less similarity to its previous version).

2.2 Disseminating the Update

To distribute the new code onto remote sensors, the update is summarized in an edit script (U in Figure 2) and then transmitted over the WSN. Such an edit script usually contains several simple update primitives such as copy, insert, replace, and remove.

Network protocol Deluge [Hui and Culler 2004] is used to disseminate the scripts to the network. The script is usually divided into a sequence of data packets. These packets may be encrypted and/or authenticated for

Fig. 4. Allocating different registers for improved energy efficiency.

security protection [Lanigan et al. 2006; Dutta et al. 2006]. The packets may also be grouped so that when remote sensors receive groups out of order, they are still able to perform updates independent of the receiving order.

When a sensor node receives the complete update script, it runs the script interpreter to build the new binary code based on the old version and the script.

3. UPDATE-CONSCIOUS REGISTER ALLOCATION

As mentioned in the overview, we perform UCC in the code generation stage, which typically involves register allocation, data, and code placement. We focus on the former two tasks in this article and will investigate the code placement problem in our future work. In this section, we discuss the update-conscious register allocation (UCC-RA) design.

We will first illustrate our strategy using a motivational example and formulate the update-conscious allocation problem as a mixed integer nonlinear programming problem, which targets at both performance improvement and energy minimization. It is not a linear problem due to the nonlinear specifications of the update energy consumption. We then discuss how to approximate the non-linear specification using an integer linear programming (ILP) program. The latter can be solved magnitudes times faster than the former for problems of similar sizes.

3.1 Example: Register Allocation and Code Similarity

While the register allocation problem has been well studied with great success in the past 2 decades [Chaitin et al. 1981; George and Appel 1996; Goodwin and Wilken 1996; Naik and Palsberg 2002; Zhuang and Pande 2006; Koes et al. 2006], no algorithm has been proposed to address the update problem encountered in sensor networks. In Figure 4, we illustrate why different register allocation decisions can greatly impact the code similarity and the update cost. In this example, two variables, a and b, initially have disjoint live ranges and can be allocated to the same register R1 (Figure 4(a)). Assume a small code change extends b's live range into a's. If there are enough free registers, a modern register allocator will assign different registers to them, as depicted in Figure 4(b).

Variable b is assigned to a new register $R2$, resulting in a name change for all the uses in subsequent statements in the statement range $\{5,15\}$. In contrast, an alternative update-conscious decision may allocate b to $R2$ only for the range $\{5,11\}$ where $R1$ is not free, and match the old allocation for the range $\{12, 15\}$ with one extra `mov` instruction, as shown in Figure 4(c). By comparing these two solutions, it is clear that while the solution (b) achieves better code quality, the solution (c) results in less update cost. The discrepancy in energy consumption between data transmission and instruction execution makes the solution (c) more appealing, as it consumes less energy unless the code is very frequently executed, or the update is extremely rare.

3.2 Update-Conscious Register Allocation

The basic idea of UCC-RA is to retain mostly the old register assignments and perform new register allocations to changed and new instructions, with preferences to the register assignment in the older version.

To achieve this, we assume the same optimization strategy in compiling both the old and the new code. With similar code structures, we first identify new IR instructions as “changed” or “nonchanged,” and then group successive instructions of the same type into chunks. A chunk is considered as “non-changed” if (i) all its instructions are not changed, and (ii) the chunk size is larger than K instructions, where K is a predetermined threshold to prevent from overly small chunks. Otherwise, it is merged with neighboring chunks to form a “changed” chunk.

Our register allocator then allocates registers for each changed chunk and gradually matches the register assignment or allocation decisions from both changed and nonchanged chunks for semantic correctness. Decisions for changed chunks are made by our UCC-RA, while decisions for unchanged chunks are taken from the old code before the update. The two decisions are made conjointly. If a variable’s live range spans across the chunk boundary, from “changed” to “nonchanged” or vice versa, then the assignment in the “changed” chunk gives preference to the assignment in the “nonchanged” chunk to maximize the similarity. However, this preference may not always be adopted by the allocator. If the allocator decides to use a new register in the “changed” chunk, then a `mov` instruction between the two chunks should be inserted to move data between the new and the old registers. Register preference should also be given to the same variables on different control-flow paths (they might be of different chunk types). However, if the allocator chooses a different register, then a `mov` instruction is also necessary.

Clearly, placing too many interregister movement instructions requires not only transmitting more update data to remote sensors but also executing more instructions at runtime. Therefore, it is desirable to develop a precise cost-benefit model such that an interregister movement instruction is inserted only if it is estimated to be energy efficient.

Preferred-register tag. In UCC-RA, we tag each variable in an unchanged IR instruction with the register name that was assigned in the old

Fig. 5. Decision variables.

binary. A preferred register tag is a hint to improving code similarity in UCC-RA.

3.3 Formalizing the Update-Conscious Register Allocation

Motivated by the 0/1 ILP research for register allocation [Goodwin and Wilken 1996], we formalize our UCC-RA as a nonlinear integer programming problem. We use a simple example in Figure 5 to explain our proposed procedure. The code contains several instructions: The first two are the definitions of variable a and b , respectively, while the third uses both variables. Let us assume after Statement (6), a is dead but b is still alive, and the preferred registers of a and b are $R1$ and $R2$, respectively.

The decision variables. For the code chunk in Figure 5(a), we first introduce a set of decision variables that represent the register assignments we need to make at each program point. For example, if variable a is allocated to register $R1$ at Statement (1), then we have $X_{def.a.1}^{R1} = 1$ and $\forall Ri, Ri \neq R1, X_{def.a.1}^{Ri} = 0$. Here, $X_{def.a.s}^{Ri}$ is a decision variable that shows if variable a is assigned to the register Ri at Statement s . A decision variable X_* can take value 0 or 1, with 1 meaning that the corresponding assertion is true, and 0 otherwise. As another example, if we decided to insert an instruction “mov $R2$ to $R3$ ” for b before Statement (4), we set $X_{mov.out.b.4}^{R2} = 1$, $X_{mov.in.b.4}^{R3} = 1$ and all other mov decision variables $X_{mov.*.b.4}^*$ as 0. As discussed, such a mov instruction may be inserted to release $R2$ for other variables or to match the old assignment of b to $R3$ after Statement (4). The following is a full list of decision variables that we used in UCC-RA.

$a/s/Ri$	variable a / statement s / Register Ri ($1 \leq i \leq 31$);
$X_{mov.out.a.s}^{Ri}$	if a is moved from Ri to another register at s ;
$X_{mov.in.a.s}^{Ri}$	if a is moved from another register to Ri at s ;
$X_{def.a.s}^{Ri}$	if a is allocated to Ri at its definition point s ;
$X_{cont.a.s}^{Ri}$	if a is allocated to Ri after its def point s ;
$X_{lastUse.a.s}^{Ri}$	if a is allocated to Ri at its last use point s and a is dead after s .
$X_{use.a.s}^{Ri}$	if a is allocated to Ri at s , but not in Ri after s ; statement s is not the last use.
$X_{useCont.a.s}^{Ri}$	if a is allocated to Ri at s , and is also in Ri after s ; statement s is not the last use.
$X_{st.a.s}^{Ri}$	if a is spilled from Ri to memory after s ;
$X_{ld.a.s}^{Ri}$	if a is loaded from memory to Ri before its use point s ;
$X_{cont.a.s}^{mem}$	if the variable is kept in memory after the statement s ;

When defining proper decision variables, we aim to keep their total number small so that the solver takes less time to find a solution. For example, we introduce two decision variables $X_{mov.in.a.s}^{Ri}$ and $X_{mov.out.a.s}^{Ri}$ instead of a more intuitive $X_{mov.a.s}^{Ri \leftarrow Rj}$ (move a from Rj to Ri at Statement s) because of the following reason. Assume there are 31 registers; the one-variable definition would introduce 31×30 *mov* decision variables for each variable at a program point. This will increase the problem size and slow down the solver. Instead, we decouple the *mov*'s source register from the destination register such that only 31×2 decision variables are required. Then, we simply correctly combine the corresponding move-in and move-out variables to implement the register move.

The constraints. With above decision variables, we convert the register allocation problem into a problem of assigning value 0 and 1 to these variables. To ensure that the value assignment can be mapped back to a valid register assignment, these variables are subject to a set of constraints.

We first define the constraints for variable definitions. Each variable should be allocated to one and only one register at its definition point. Thus, we have, for each variable a at its definition point s , one and only one $X_{def.a.s}^{Ri}$ can be 1, or,

$$\sum_{\forall Ri} X_{def.a.s}^{Ri} = 1. \quad (1)$$

To ensure valid interregister movements, we define constraints on *mov* decision variables as well. Since we may and may not insert a move instruction at a program point and the move-in and move-out decision variables should appear in pairs, we have:

$$\sum_{\forall Ri} X_{mov.out.a.s}^{Ri} \leq 1$$

$$\sum_{\forall Ri} X_{mov.out.a.s}^{Ri} = \sum_{\forall Ri} X_{mov.in.a.s}^{Ri} \quad (2)$$

At a Statement s , variable a may be loaded from the memory, or come from interregister movement. After defining the variable, the value in the register may be spilled to the memory, or moved to another register, or stay for later

use. Thus, we have:

$$\begin{aligned} X_{st.a.s}^{Ri} &\leq X_{def.a.s}^{Ri} + X_{mov.in.a.s}^{Ri} \\ X_{mov.out.a.s}^{Ri} &\leq X_{def.a.s}^{Ri} \\ X_{cont.a.s}^{Ri} &\leq X_{def.a.s}^{Ri} + X_{mov.in.a.s}^{Ri}. \end{aligned} \quad (3)$$

For the code spill at a definition point, only a store instruction may be possibly generated. Thus, we have:

$$X_{cont.a.s}^{mem} \leq \sum_{\forall Ri} X_{st.a.s}^{Ri}. \quad (4)$$

We next define the constraints for variable uses. Since we can know if a use is the last use (through backward analysis), $X_{lastUse.a.s}^{Ri}$ is always exclusive from $(X_{use.a.s}^{Ri} + X_{useCont.a.s}^{Ri})$. In addition, $X_{use.a.s}^{Ri}$ and $X_{useCont.a.s}^{Ri}$ are exclusive, and a use should be in a register. These are specified as:

$$\begin{aligned} \sum_{\forall Ri} X_{lastUse.a.s}^{Ri} &= 1; \quad or \\ \sum_{\forall Ri} (X_{use.a.s}^{Ri} + X_{useCont.a.s}^{Ri}) &= 1; \end{aligned} \quad (5)$$

At a use point, a variable may be located in a register due to its use in the previous instruction, loaded from the memory, or moved from another register. Depending on whether it is the last use, we have one of the following two constraints:

$$\begin{aligned} X_{use.a.s}^{Ri} + X_{useCont.a.s}^{Ri} &\leq X_{cont.a.(s-1)}^{Ri} + X_{ld.a.s}^{Ri} + X_{mov.in.a.s}^{Ri} \\ X_{last.a.s}^{Ri} &\leq X_{cont.a.(s-1)}^{Ri} + X_{ld.a.s}^{Ri} + X_{mov.in.a.s}^{Ri} \end{aligned} \quad (6)$$

Since we only generate load spill or interregister movement before the use point, we have:

$$\begin{aligned} \sum_{\forall Ri} X_{ld.a.s}^{Ri} &\leq X_{cont.a.(s-1)}^{mem} \\ \sum_{\forall Ri} X_{mov.out.a.s}^{Ri} &\leq X_{cont.a.(s-1)}^{Ri} \end{aligned} \quad (7)$$

The following constraints are used to ensure that one register is assigned to only one variable at a time. Next, we introduce two temporary variables $U_{var.s}^{Ri}$ and $V_{var.last.s}^{Ri}$. They are not new decision variable. Instead, they are introduced only to facilitate our discussion in the article, and are replaced by corresponding decision variables in the implementation.

$U_{a.s}^{Ri}$ is defined to describe the register assignment to the variable a at instruction s .

$$U_{a.s}^{Ri} = \begin{cases} X_{def.a.s}^{Ri} & : \text{ if } a \text{ is defined at } s \\ X_{use.a.s}^{Ri} + X_{useCont.a.s}^{Ri} & : \text{ if } a \text{ is used at } s \\ X_{lastUse.a.s}^{Ri} & : \text{ if } a \text{ is used at } s \text{ and no longer used in} \\ & \text{the later instructions} \end{cases} \quad (8)$$

$V_{a.last_s}^{Ri}$ is defined to show the register assignment information of the last access point of variable a before instruction s . Depending on whether instruction $last_s$ is a definition point or use point of variable a , $V_{a.last_s}^{Ri}$ is calculated as follows.

$$V_{a.last_s}^{Ri} = \begin{cases} X_{cont.a.last_s}^{Ri} & : \text{ if } a \text{ is defined at } last_s \\ X_{useCont.a.last_s}^{Ri} & : \text{ if } a \text{ is used at } last_s \end{cases} \quad (9)$$

To make sure that there is no register assignment conflict between the current variable and the active variables that are processed before, the following constraint is applied:

$$\sum_{\forall var} V_{var.last_s}^{Ri} + U_{a.s}^{Ri} \leq 1 \quad (10)$$

For example, the following equations show the constraints at Statement (2) in Figure 5:

$$\begin{aligned} U_{b.2}^{Ri} &= X_{def.b.2}^{Ri} \\ V_{a.last_use}^{Ri} &= X_{cont.a.1}^{Ri} \\ \text{thus, } X_{def.b.2}^{Ri} + X_{cont.a.1}^{Ri} &\leq 1 \end{aligned} \quad (11)$$

Also, in order to avoid the register assignment conflict between the variables, which are used in the same instruction, the following constraint needs to be applied:

$$\sum_{\forall var} U_{var.s}^{Ri} \leq 1 \quad (12)$$

For example, the following constraints at Statement (6) in Figure 5:

$$\begin{aligned} U_{a.6}^{Ri} &= X_{lastUse.a.6}^{Ri} \\ U_{b.6}^{Ri} &= X_{use.b.6}^{Ri} + X_{useCont.b.6}^{Ri} \\ \text{thus, } X_{lastUse.a.6}^{Ri} + X_{use.b.6}^{Ri} + X_{useCont.b.6}^{Ri} &\leq 1 \end{aligned} \quad (13)$$

For Mica2 micro controllers, we need to enforce another type of constraint. Each register in Mica2 has 8 bits (i.e., 1 byte). A 32-bit integer variable should be allocated to four consecutive registers (i.e., byte a , $a+1$, $a+2$, and $a+3$) should be in register Ri , $Ri+1$, $Ri+2$, and $Ri+3$, respectively:

$$\begin{aligned} X_{use.(a).s}^{Ri} &= X_{use.(a+1).s}^{Ri+1} \\ X_{use.(a+1).s}^{Ri} &= X_{use.(a+2).s}^{Ri+1} \\ X_{use.(a+2).s}^{Ri} &= X_{use.(a+3).s}^{Ri+1} \end{aligned} \quad (14)$$

At the boundary of changed and unchanged code chunks, and at the merge point of control flows, we insert interregister move instructions to make sure that the values are in proper registers before their next uses. In our future work, instead of performing interregister movements, we will introduce constraints similar to those in Goodwin and Wilken [1996] for the merge point of control flows.

$$E_{total} = E_{changed_IR} + E_{unchanged_IR} + E_{spill} + E_{extra} \quad (15)$$

where

$$E_{changed_IR} = \sum_{\forall s} (chg(s) \times freq(s) \times E_{exe}) + \sum_{\forall s} (chg(s) \times E_{trans}) \quad (16)$$

$$E_{unchanged_IR} = \sum_{\forall s} ((1 - chg(s)) \times freq(s) \times E_{exe}) + \sum_{\forall s} ((1 - chg(s)) \times (1 - \prod_{\forall a} X_{def/use.a.s}^{prefer(a,s)}) \times E_{trans}) \quad (17)$$

$$E_{spill} = \sum_{\forall s, a, Ri} (freq(s) \times (X_{st.a.s}^{Ri} + X_{ld.a.s}^{Ri}) \times E_{exe}) + \sum_{\forall s, a, Ri} ((1 - spill(a, Ri, s)) \times (X_{ld.a.s}^{Ri} + X_{st.a.s}^{Ri}) \times E_{trans}) \quad (18)$$

$$E_{extra} = \sum_{\forall s, a, Ri} (freq(s) \times X_{mov.in.a.s}^{Ri} \times E_{exe}) + \sum_{\forall a, s, Ri} (X_{mov.in.a.s}^{Ri} \times E_{trans}) \quad (19)$$

Fig. 6. The objective function.

The objective function. The goal of our integer programming is to minimize the objective function on total energy consumption, as expressed in Equation (15) in Figure 6. The equation defines the total energy consumption of the changed IR chunk under different register allocation decisions. The notations used in Equation (15) are listed as follows. Other terms are explained as follows.

E_{trans}	the energy consumed to disseminate one instruction in WSN;
E_{exe}	the energy consumed to execute one instruction. We use the average number here and differentiate the memory access (load, store) and ALU instructions in the implementation.
$prefer(a, s)$	the preferred register for variable a at statement s;
$freq(s)$	the execution frequency count of statement s;
$chg(s)$	if s is an unchanged IR instruction. $chg(s)=1$ if s has been changed; =0 otherwise;
$spill(a, Ri, s)$	if variable a was spilled to Ri/loaded back from Ri at statement s in the old binary;

E_{spill} specifies the energy consumption due to code spill. It includes two components: the execution energy and the dissemination energy. The former has to do with the code quality, which is the main goal of many existing allocators. The latter is not negligible when a new spill is generated or an old spill is removed. It is zero for all other cases, that is, either $(1 - spill(a, Ri, s)) = 0$ or $(X_{ld.a.s}^{Ri} + X_{st.a.s}^{Ri}) = 0$ in Equation (13). For example, if a is spilled to R1 in both new and old binaries, then we have zero transmission cost:

$$\begin{aligned} &\text{for R1, } 1 - spill(a, R1, s) = 0, X_{ld.a.s}^{R1} + X_{st.a.s}^{R1} = 1 \\ &\text{for Ri (Ri} \neq R1), 1 - spill(a, Ri, s) = 1, X_{ld.a.s}^{Ri} + X_{st.a.s}^{Ri} = 0 \end{aligned}$$

$E_{changed_IR}$ specifies the energy consumption due to changed IR instructions. It includes both the execution and the dissemination energy consumption as well. As we can see, no matter which register allocator is used, a changed IR instruction always results in a binary instruction that should be disseminated to remote sensors. Therefore, $E_{changed_IR}$ is a constant in the model.

$E_{unchanged_IR}$ specifies the energy consumption due to unchanged IR instructions. Assume we have an unchanged IR instruction “a=a+b” and a and b’s preferred registers are R1 and R2, respectively. If the new allocation decision follows the old allocation scheme, then there is no dissemination cost, that is, the same binary instruction “add R1, R2” is generated. If a is assigned to a different register, say R3, and we generate “add R3, R2,” then this new instruction needs to be disseminated to replace the old one on the sensor. As shown in Equation (12), this component is nonlinear—one E_{trans} is introduced for either one or two changes of the two preferred registers.

E_{extra} is the extra energy consumption due to inserted interregister movements. This term is zero if a traditional compiler decision is used. Our UCC-RA targets at achieving overall energy efficiency, that is, E_{extra} is positive only when we can gain more reduction from other components (e.g., $E_{unchanged_IR}$).

In the previously described model, X^* are decision variables that need to be determined by the UCC-RA, while others such as $chg(s)$, $freq(s)$, and the like are known for a given code chunk. Since Equation (12) is nonlinear, the previously described formulation of UCC-RA results in a mixed integer nonlinear programming problem (MINLP) [Bonmin]. While the speed of MINLP solvers has been improved greatly in recent years [Bonmin], it is still much slower than solving a linear problem. Our experiments results show that MINLP can be orders of magnitude slower than a linear problem of similar sizes (i.e., similar number of decision variables and constraint). Next, we discuss how to convert the MINLP problem to an ILP problem through approximation.

3.4 Solving an ILP Problem

In this section, we model the update energy consumption linearly such that the UCC-RA can be solved using an ILP solver.

For an unchanged IR instruction with two variables a and b (to comply with Mica2 AVR ISA, each IR instruction in our model has at most two different operands). Assume their preferred registers are R1 and R2, respectively, we model the energy consumption as

$$\sum_{\forall s} ((1 - chg(s)) \times ((1 - X_{use.a...}^{R1}) + (1 - X_{use.b...}^{R2}))) \times E_{trans} \times \delta \quad (20)$$

where $\delta = 3/4$, a coefficient that approximates the update cost. It is decided as follows. Assume each variable has equal opportunity of being assigned and not assigned to its preferred register. For the instruction with two variables a and b and preferred registers R1 and R2, respectively, there are four possibilities altogether: (i) a is in R1, b is in R2; (ii) a is in R1, b is not in R2; (iii) a is not in R1, b is in R2; (iv) a is not in R1, b is not in R2. It is clear that Case (i) has no updated cost, while each of other three cases needs to update one instruction. Therefore, the average update cost is $(3/4) \times Cost_{single}$, which decides δ to be $3/4$.

Source:	Assembly:	Source:	Assembly:	Update script:	Source:	Assembly:	Update script:
uint_16 a; uint_16 b; uint_16 c; ...	; a offset=0 ; b offset=2 ; c offset=4 ...	uint_16 a; uint_16 b; uint_16 c; uint_16 d; ...	; b offset=0 ; c offset=2 ; d offset=4 ...	[R: ld ...] [R: st ...] [R: lsl ...]	uint_16 a; uint_16 b; uint_16 c; uint_16 d; ...	; d offset=0 ; b offset=2 ; c offset=4 ...	[R: lsl ...]
a=100; c = a + b; ...	li r1, 100 ld r2, 0xa02 add r2, r2, r1 st r2, 0xa04	a=100; c = 100 + b; d = b <<1;	li r1, 100 ld r2, 0xa00 add r2, r2, r1 st r2, 0xa02 lsl r2		a=100; c = 100 + b; d = b <<1;	li r1, 100 ld r2, 0xa02 add r2, r2, r1 st r2, 0xa04 lsl r2	
(a)		(b)			(c)		

Fig. 7. An incremental data allocation example ((a) original source and assembly code; (b) new code and the update script; (c) incrementally generated new code with a smaller update script).

After converting the model into an ILP problem, we adopt a widely used ILP solver—LP_solve [Berkelaar 2006]—to find the optimal assignment of decision variables such that the cost (modeled in the objective cost function) is minimized. We then map decision variables back to register assignments, and generate the code and the corresponding update script as well.

4. UPDATE-CONSCIOUS DATA ALLOCATION

In addition to register allocation schemes, the data allocation strategy can also affect the similarity between different versions of code, as illustrated in the example in Figure 7. In the original code (Figure 7(a)), three variables *a*, *b*, and *c* are allocated with offset 0, 2, and 4, respectively, to a base address. Assume the code is updated by replacing variable *a* with a constant, and introducing a new variable *d*. The existing compiler may generate the data allocation scheme, as shown in Figure 7(b), in which all variables are assigned with new offsets, resulting in three update primitives in the update script. However, an update-conscious algorithm should put the new variable *d* in *a*'s location, as shown in Figure 7(c), resulting in only one update primitive in the script. On the other hand, if there was no *d* in the new code and if we did not reclaim the word taken by *a*, we would waste the word in the data segment or more if the function is recursively invoked. This will increase the memory footprint on remote sensors.

4.1 Threshold-Based Data Allocation

To address the problem described previously, we propose a threshold-based data allocation mechanism. The intuition is to reuse the space of deleted variables as much as we can. If there are more new variables than deleted variables, we will first use up the space of the deleted variables and then allocate more space. If there are more deleted variables, then we have two options to choose from: (i) relocate some old variables and (ii) do not relocate. The first option does not waste the space resource on sensor node, but it needs to change the program code because of the relocated variables. The second option incurs less code changes but leaves “holes” in the data segments at runtime. As a hybrid of these two options, our proposed algorithm ensures that the total wasted space is less than a given threshold—*SpaceT*. For ease of illustration, we elaborate

on the procedures for variables of word type only. The principle can be applied to other data types such as array and composite structures similarly.

First, we collect the following profiles for each function $P_i (i \geq 0)$ in the program. P_0 is a dummy function that contains all global variables.

$DelV_i$	the total number of deleted variables;
$NewV_i$	the total number of new variables;
$Depth_i$	the projected maximal simultaneous instances of P_i ;
$Usage_i(a)$	the usage of variable a in P_i .

Second, we gradually allocate new variables within each procedure P_i as follows. We do not remove the deleted variables directly. Instead, we only mark them as deleted variables so that their space can be reused by new variables. If $NewV_i \geq DelV_i$, we reuse all the space from deleted variables and allocate extra space to satisfy the remaining new variables. If $NewV_i < DelV_i$ (i.e., new variables cannot reuse all space of the deleted ones), then we compute that there are $Extra_i = Del_i - NewV_i$ number of words left to be filled and move to the next step.

Third, we adjust the data allocation by incrementally relocating the last variable in each function. We keep moving the last variable into a “hole” left by a deleted variable, until all the holes are filled. That is,

$$\sum_{\forall P_i} Extra_i \times Depth_i \leq SpaceT \quad (21)$$

As shown in Figure 7, such a move will cause changes in all the instructions that use the last variable. If Equation (21) cannot be satisfied for all procedures, to keep the changes minimum, we should first serve those that might demand the most runtime memory but have the least number of uses. That is, we should find a procedure j such that

$$\frac{Depth_j}{Usage_j(last)} = MAX \left(\frac{Depth_i}{Usage_i(last)} \right) \quad (\forall i, Extra_i > 0) \quad (22)$$

We then relocate the last variable in procedure j to one deleted memory word. By doing so, we can shrink the maximal runtime memory usage by $Depth_j$ (as it is the last variable in that procedure), and incur less code changes (as the variable with less usage is selected). We then decrement $Extra_j$ and continue this step until Equation (21) is satisfied.

For example, in Figure 7, if d is not introduced, we will reuse a ’s space with c if $SpaceT = 0$ (i.e., no wasted space). This will result in an edit script with two primitives to update c and d , respectively. This code still outperforms the default scheme in Figure 7(b), which requires three update primitives.

The data allocation problem may become more complicated if it is coupled with code generation where data offset are encoded with instruction types. For example, successive instructions using the postincrement addressing (PIA) mode will access successive data in memory with implicit address increment between two instructions. If data is relocated, new instructions must be inserted to change the memory access address in the next instruction. Fortunately, we experimented with the gcc 3.4.3 compiler and found that the PIA mode is

primitive	format and operation	cost (bytes)
insert	 number of bytes to be inserted	1 + number
replace	 number of bytes to be replaced	1 + number
copy	 number of bytes to be copied	1
remove	 number of bytes to be removed	1
shift	 [A1, A2, S] 3 words: shift S bytes from A1 to A2	7
clone	 2 words 1 word * number of register mappings [A1,A2,Ra1...Ra32,Rb1...Rb32] clone instructions [A1,A2]. Replace all Rai with Rbi.	5 + 2*number

Fig. 8. The update primitives.

mainly used to access the four bytes of an integer variable and thus is insensitive to the variable relocation. For this reason, we do not consider the impact the PIA mode when relocating the data. If they are used beyond a word boundary, we treat them individually by inserting new addressing instructions.

5. UPDATE DISSEMINATION

With the new binary code generated by the UCC technique, we summarize the code difference into a small update script and then employ Deluge [Hui and Culler 2004], the dissemination protocol in TinyOS to disseminate the script to the network. After receiving the script, each sensor node rebuilds the new binary code from the script and the old code.

5.1 Script Generation

An update script consists of a sequence of update primitives that guides how to construct the new code from the old version. Clearly, the design of script primitives affects the size of update script and thus the amount of data to be transmitted. In this article, we adopt six different types of primitives, as shown in Figure 8. The first five are similar to those in Reijers and Langendoen [2003].

5.1.1 Simple Primitives. There are four simple update primitives—insert, replace, copy, and remove. Both insert and replace primitives have 1-byte opcode and n bytes of data/instructions to be incorporated. The copy and remove primitives take 1 byte each and specify the size of old data/instruction block to be copied or removed.

Figure 9 shows a simple example of the basic primitives. The new version contains three chunks of code, [100, 110], [112, 130], and [132, 140]. Both the first and third chunks can be found in the old code, while the second chunk is new. Therefore, the update script contains two copy primitives and one insert

Fig. 9. An example of update script involving basic primitives. New code [112,130] is inserted. [112,120] in the original code is now moved to [130,140] in the new version.

primitive. The insert primitive has a 1-byte opcode and 10 instructions (or 20 bytes). The total size of the update script is 23 bytes.

5.1.2 Shift Primitive. Next, we describe the shift primitive [Reijers and Langendoen 2003], which is used to adjust the branch/jump instructions' destination addresses due to shifted instruction/data blocks.

As Figure 8 shows, the shift primitive contains a 1-byte opcode and another three words to indicate that the code segment $[A1, A2]$ is now moved to $[A1+S, A2+S]$. Thus, all the branch/jump instructions whose destination addresses are in the range $[A1, A2]$ need to shift the destination addresses by S .

To interpret the shift script, the binary rewriter running on the sensor nodes needs to have simple decoding ability—it picks up each control transfer instruction, checks whether its target address falls in the range that needs to be shifted, and updates them with new addresses. For absolute branch instructions, their target addresses can be decoded from the instruction directly; for relative branch instructions, their target addresses can be computed by adding the relative offset to the address of the current instruction.

Figure 10 shows an example using the shift primitive. Due to the insertion of new code, the chunk $[112, 120]$ in the old version is now moved to $[132, 140]$ in the new version. Thus, all the control-flow instructions that jump to any instruction in this chunk need to be updated. In the example, the shift primitive specifies that all the branch instructions whose targets are in the address range $[112, 120]$ should be shifted by 20.

When one block movement causes several changes in the code, using shift primitive helps to minimize the script size. The trade-off is that a slightly more powerful interpreter has to be installed on the sensor side such that it can decode each instruction type to extract the desired target address.

5.1.3 Clone Primitive. When we study the real applications in TinyOS, we observed that inline functions were widely used to improve the code readability while reducing the runtime overhead. We found that a small source code level

Fig. 10. An example of update script involving shift primitive. New code [112,130] is inserted. [112,120] in the old code is now moved to [132,140] in the new version. Some control-flow instructions are affected due to this address change.

change to an inline function might cause binary changes at all the places where the inline function is called. The differences are similar to each other, since they are compiled from the same source code. In most cases, the only difference is the register assignment, due to different sets of free registers at different places.

Base on this observation, we introduced the `clone` primitive. Assuming that an inline function is called at multiple places, such as block [A1,A2] and [B1,B2], where A1, A2, B1, and B2 are the instruction addresses. The code structures of these two blocks are exactly the same because they have the same source code; however, the register assignments are different due to the context differences. If we use block [A1,A2] as the comparison base, try to match the register allocation between these two blocks, we can get the following register mapping between them, $(R_{a1}, R_{a2}, \dots, R_{an}) \Rightarrow (R'_{b1}, R'_{b2}, \dots, R'_{bn})$. Given such information, instead of using a sequence of the other primitives to describe the updated/new code of block [B1,B2], we could rather copy instructions from block [A1,A2], and slightly change the register assignments according to the register mapping to rebuild block [B1,B2].

The `clone` primitive has a 1-byte opcode, several bytes to specify the starting and ending addresses of the code segment where the code would be copied from, and the register mapping (Figure 8). The primitive length varies according to the register mapping complexity. Assume there are N pairs of register mappings, the `clone` primitive length is $5+2*N$.

To interpret the `clone` primitive, the sensor node will copy the instructions from code segment [A1,A2] and replace all (R_{ai}) with (R_{bi}) .

Figure 11 shows an example using the `clone` primitive. Both the code [200,206] and [100,106] are compiled from the same inline function. Instead of generating the update script for [200,206] by using the `insert` primitive, the `clone` primitive is used to specify that the second code block clones the block [100,106] while registers r_1 and r_2 needs to be updated to be r_{11} and r_{12} , respectively.

The `clone` primitive can reduce the script size when the inline function is called at multiple places, and the register mapping is clear. However, if the

Fig. 11. An example of update script involving clone primitive. New code [200,206], which is compiled from the same inline function as code [100,106], is inserted.

register mapping is too complicated, the script size could be very big, in which case it is better to use basic primitives, such as insert and replace. In addition, it requires that the sensor-side interpreter has simple decoding ability to extract register names from different instruction types and replace them with new ones.

5.2 Script Dissemination

With the script primitives defined previously, we can generate the code update script. In practice, the script is divided into a sequence of data packets with each packet having 23 bytes payload in TinyOS. For security protection reasons, encryption and/or authentication can be applied to these data packets [Lanigan et al. 2006; Dutta et al. 2006]. There are a few dissemination protocols developed to distribute the software update in WSNs, such as Deluge [Hui and Culler 2004] and MNP [Kulkarni et al. 2005]. We chose to use the Deluge protocol to do the code dissemination, because it is a reliable data dissemination protocol for propagating large data objects from one source node to many nodes, which are multihops away from the sink in WSNs.

To improve dissemination effectiveness in lossy wireless communication, Deluge groups the packets into a number of pages with each page containing a fixed number of packets. Pages are disseminated sequentially, while the packets within one page may be received out of order. For simplicity, in Deluge, one sensor needs to receive all packets within one page before receiving the next page.

Next, we briefly describe Deluge protocol to assist understanding our experimental results while more details can be found in Hui and Culler [2004]. Deluge uses three types of messages to disseminate the code. Each sensor node periodically broadcasts the advertisement message (ADV), which contains the information of the code to be disseminated, including the number of pages it has received so far. The states of nodes are set to be Maintain at the beginning. When a sensor node *S* receives an advertisement, which indicates that the neighbor *N* has finished more pages than itself, it will send out a request message (REQ) to *N* asking for a new page, and change its own state to Request. The state is changed back after receiving all packets of the requested page. Node *N*

Fig. 12. Advertise-request-data handshaking protocol in Deluge.

changes its state to Transmit when it receives the request message, and starts sending the requested packets in DATA messages. After the transmission, its state is changed back to Maintain state. Figure 12 illustrates the ADV-REQ-DATA handshaking protocol in Deluge [Hui and Culler 2004].

5.3 Reconstructing the New Code at the Sensor Side

In Deluge, the received code is placed in the data memory and is written to the program flash after passing CRC and security checks. In our augmented design, the sensor needs to reconstruct the new code—a simple sensor-side interpreter retrieves both the old code and the update script, incrementally reconstructs the new code segments in the memory, and writes the new code segments to the program flash.

6. EXPERIMENTS

We have implemented our proposed UCC-RA and data allocation (UCC-DA) techniques, and compared them with the results generated by the GNC C compiler (GCC-RA and GCC-DA). In this section, we discuss our experimental settings and present the results on code quality, energy efficiency, and compilation time.

6.1 Settings

We simulated a sensor network that consists of Mica2 mote nodes [Xbow] running TinyOS [2008], an open-source operating system designed for WSNs. The processor that Mica2 (MPR400CB model) uses is the AMTEL AVR micro controller—ATmega128L [Xbow].

To compile the code for Mica2, we chose `ncc`, the NesC compiler included in TinyOS release, and `avr-gcc`, the GNU C compiler (GCC) re-targeted for AMTEL AVR microcontrollers. We used the `-O3` option to compile the code and ensure the code fit in the sensor storage (i.e., we considered the `-Os` option as well). We used the default register allocator of the `gcc/avr-gcc`, for using the new iterative graph allocator (with the option `-fnew-ra`) would give similar results.

We selected Avrora, an instruction-level sensor network simulator, to collect the execution cycles of the code before and after compiling the updated code with UCC and GCC (the accuracy of the simulator has been reported in prior work [Titzer et al. 2005]). We then integrated the energy model and execution profiles to study the energy consumption trade-offs with different compilation approaches. The update script generator is built over Diffutils tool [Diffutils].

Benchmark	Source	Details
Blink	TinyOS	It starts a 1Hz timer and toggles the red LED every time it fires.
CntToLeds	TinyOS	It maintains a counter on a 4Hz timer and displays the lowest three bits of the counter value. The red LED is the least significant of the bits, while the yellow is the most significant.
CntToRfm	TinyOS	It maintains a counter on a 4Hz timer and sends out the value of the counter in an IntMsg AM packet on each increment.
CntToLeds AndRfm	TinyOS	It maintains a counter on a 4Hz timer; it combines the tasks performed by CntToRfm and CntToLeds.
AES	Crypto Lib	It encrypts a given 128-bit input buffer using AES algorithm. We select the encryption code in the experiment.
Deluge	TinyOS	The mulithop code dissemination protocol in TinyOS. We tracked its continuous updates in different TinyOS versions as a real-life case study.

Fig. 13. Benchmark programs.

6.2 Code Update Benchmark

Applications running on remote sensors may be updated for various reasons, for example, bug fixes, code patches, sensor reconfigurations for adapting to changing environments, and change of applications. A recent study showed that code fixes and sensor reconfigurations happen more often than changing the application completely because the latter is much more costly [Dunkels et al. 2006].

We categorize different updates into three types according to their impact on code structures: (i) small changes, which are made to local basic blocks; (ii) medium changes, which include changes in a large function or across several functions, but still preserve the overall structure of the original code; (iii) large changes, which significantly change the code structure. Frequent updates, such as code fixes and sensor reconfigurations, are mainly small or medium changes, while replacing the application with a new one introduces medium to large changes. We show our results for all three types of changes in this section.

The benchmark programs that we used for testing our UCC-RA and UCC-DA are listed in Figure 13. Those are from the TinyOS release and the crypto library [Dai 2007]. In particular, the Deluge program is the code dissemination protocol that we used to disseminate the update script. The implementation undergoes several bug fixes and updates in consecutive TinyOS versions. We tracked these updates as a real-life case study and reported the results in Section 6.8.

6.3 The Dissemination Cost

Figure 14 summarizes the synthetic updates that we made to the benchmarks. The updates vary from small, through medium, to large changes, as described in the following text.

—The small and medium test cases cover a wide range of changes including constant changes, variable changes, parameter changes, instruction changes,

Case #	Update Level	Update details
1	Small	In CntToLeds: Change the color of blink.
2	Small	In Blink: Insert one local variable and one use in run_next_task.
3	Small	In AES: Insert one local variable and use it within the loop in aes.encrypt.
4	Small	In AES: Change one instruction in aes.encrypt.
5	Small	In AES: Insert a local variable in aes.encrypt and use it twice — within and outside the loop.
6	Small	In Blink: Add a new parameter in TOSHrun_task.
7	Medium	In CntToLeds: Insert three variables and their uses;
8	Medium	In CntToRfm: Insert a global variable and use in three different functions.
9	Medium	In CntToRfm: Insert a local variable and use it several times in TOSH_run_next_task function.
10	Medium	In Blink: Insert a global variable and use it in a new if/then branch in TOSH_run_next_task function.
11	Medium	In Blink: Add an else branch for an if statement in TimerHandleFire.
12	Large	Change the application from CntToRfms to CntToLedsRfm
13	Large	Change the application from CntToLeds to CntToRfms.

Fig. 14. Experimental update details.

and control-flow changes. More complex updates may require one or more such changes.

- Complex updates tend to create changes over many functions, though most of these test cases impact only one function. To fairly evaluate the UCC-RA and decouple its impact from data allocation and code layout, we only report the changes in the functions that are directly affected (rather than, for instance, code shifting due to expansion/shrinkage of neighbor functions). In addition, we observed minimal interprocedural correlation. For example, the same global variable can be assigned with different registers in different functions. Therefore, the overall impact of large updates can be estimated by summarizing the changes in simple updates.
- We evaluate the code changes using $Diff_{script_size}$, the size of the update scripts that are used to change the old binaries to the new ones.

We first conducted experiments to compare the dissemination cost between UCC-RA and GCC-RA. For GCC-RA, we manually find the best match between the new and the old binaries. This is the lower bound of existing binary-diff-based code dissemination algorithms [Platen and Eker 2006; Reijers and Langendoen 2003]. That is, we compared our results against the best possible implementation of existing update-unconscious approaches [Platen and Eker 2006; Reijers and Langendoen 2003].

Figure 15 shows the results, in $Diff_{script_size}$, for Update Test Cases 1 to 12. As we can see, UCC-RA greatly reduces the code difference as it effectively localizes the code changes—the majority of the code can be kept the same. On the contrary, GCC-RA may generate only local changes (Test Case 1), but may also propagate local changes to a much larger range (Test Case 4).

We then studied the two large changes. Test Case 12 introduces several new functions, most of which are small inline functions. They disturb the register selection in a large function and introduce significant number of differences,

Fig. 15. The code dissemination cost.

which are seen when using GCC-RA. Fortunately, those differences are minimized in UCC-RA. Test Case 13 represents another type of large changes, the application CntToLeds is quite different from CntToRfms. The former has 828 instructions, while the latter has 4,351 instructions. It is an expensive update, since all new instructions and functions have to be disseminated across the network. There is some code similarity due to the fact that applications in the same TinyOS environment follow a generic structure. GCC-RA can reuse 422 instructions and needs to update 3,929 instructions. UCC-RA can reuse 63 more instructions, which represents an increase of 15% from GCC-RA, and accounts for about 7.6% of the old code (CntToLeds).

6.4 The Code Quality Comparison

Next, we compared the code quality resulting from different algorithms. The code quality is quantified using $Diff_{cycle}$, the changes in execution cycles between the old and new version of the binary. This metric also indicates the slowdown in execution time after applying UCC.

Figure 16 shows the results for Test Cases 1 to 12. In most of these cases, UCC-RA and GCC-RA have the same $Diff_{cycle}$ (i.e., they have the same code quality). This is because both of them can find free registers to use, and no extra spill codes need to be generated. Thus, register conflicts are small. In some cases, for example, Test Case 12, UCC-RA inserts three mov instructions, since, by doing so, it can save 406 instruction updates and achieve overall energy efficiency.

The slowdown from applying UCC-RA is negligible in nearly all cases. For example, the three cycles introduced by UCC-RA in Test Case 12 accounts for less than 0.01% of 244K cycles—the total number of cycles per single run for the application CntToRfm. In the next section, we study its energy consumption over a long period after many invocations.

Fig. 16. The performance comparison (single run).

For Test Case 13, UCC-RA only uses the preferred register tag as hint when selecting registers. It has the same code quality as the one generated by GCC-RA.

6.5 The Energy Consumption

The energy savings per update are calculated as follows. We first compute $Diff_{energy}$ (defined later in the text), the energy consumption difference (per single run) before and after the code update. It incorporates the energy consumed in both data transmission and instruction execution. Second, we compute the energy savings per update for GCC-RA and UCC-RA respectively.

$$Diff_{energy} = (Diff_{script_size} \times E_{trans} + Diff_{cycle} \times E_{exe} \times Cnt) \quad (23)$$

$$EnergySavings = Diff_{energy}^{GCC-RA} - Diff_{energy}^{UCC-RA} \quad (24)$$

where Cnt is the total number of times that the code may be executed before it retires. A code retires when either it is overwritten by a later update or the sensor node has consumed all its battery energy and dies.

Figure 17 plots the the energy savings of UCC-RA over GCC-RA as a function of Cnt , which is projected from the execution profiles and the code update frequency. Code fragments that reside in a loop or retire after a long time have larger $Cnts$ than others. From the figure, we can see that when UCC-RA and GCC-RA generate the same quality code (same $Diff_{cycle}$, such as for Test Case 1), the energy savings are independent of Cnt . The savings mainly come from the reduced transmission energy. The larger number of instructions we reduce from GCC-RA, the less data we need to transmit and the more savings we gain from UCC-RA.

When the code generated from UCC-RA runs slightly slower than that from GCC-RA (e.g., Test Case 12), extra energy will be consumed in instruction

Fig. 17. The energy savings per update with different code execution frequency.

execution. This can diminish the transmission energy savings when the code is executed very frequently. Therefore, UCC-RA adaptively inserts `mov` instructions according to execution profiles and update frequency. A large *Cnt* would disable the insertion such that UCC-RA and GCC-RA have the same energy consumption in the worst case. For example, UCC-RA falls back to take the GCC-RA's decision when the modified in code Test Case 12 is projected to execute more than 10^7 times.

6.6 The Problem Complexity and Compilation Time

Since the ILP problem is more complex to solve when the number of instructions and variables increase, we discuss the problem complexity in this section. Figure 18 plots the number of constraints as a function of instruction number. We can see that the number of constraints increases almost linearly with the number of IR instructions. We plot the number of iterations that the LP_solve [Berkelaar 2006] requires as a function of (the number of variables $\times$ the number of IR instructions) in Figure 19.

An interesting observation we found is that the preferred register tag helps to improve the performance. Comparing to an ILP-based register allocator, which allocates register from scratch, the preferred register tag is a hint to the solver and can reduce the number of iterations that solver needs to try. As an extreme case, we also tested misleading preferred register tags (e.g., variables are assigned to the preferred register tag randomly), and we found the solver may need two or three times more iterations to solve.

Fig. 18. The number of constraints as a function of number of IR instruction.

Fig. 19. The number of iterations as a function of the number of variables $\times$ the number of IR instructions.

To see how fast the problem can be solved, we conducted timing experiments on Intel Xeon 3.6GHz processor running Fedora Linux 2.4.21 kernel. The physical memory size is 2GB, while in the experiments, the largest observed memory usage is less than 256MB. Figure 20 shows that the average time required to solve one iteration increase about linearly with the problem complexity. It

Fig. 20. The time to solve one iteration as a function of the number of variables $\times$ the number of IR instructions.

Case #	Update details
D1	In CntToRfm: Insert several global variables.
D2	In CntToLEDs: Shuffle the order of variables and change variable names.

Fig. 21. Data layout update details.

usually takes the solver less than 175 seconds to allocate registers for a chunk of 250 IR instructions. As a comparison, it takes GCC-RA less than 1 second to solve the same problem. While UCC-RA is much slower than GCC-RA, it is not a significant problem for WSNs due to the following reasons: (i) sensor applications are small programs limited by the memory size of the sensor node, (ii) UCC-RA is applied only to the identified changed chunks instead of the complete functions or the whole application, (iii) it is worthwhile to trade the compilation time at the server side, where both energy and computation power are abundant, for the energy savings on sensor nodes where resources are highly constrained.

We also performed experiments on testing whether approximating the original nonlinear integer programming problem with a linear problem degraded the final results. We observed the same allocation decisions for all the test cases with or without the approximation. The only difference is that solving nonlinear problems is orders of magnitude slower than a linear problem.

6.7 The Update-Conscious Data Allocation

Next, we studied the effectiveness of UCC-DA. When new global variables are added to a program (Test Case D1 in Figure 21), the data layout could change

Case #	Versions	Update Level	Update details
1	1.52 $\Rightarrow$ 1.53	Small	Add one statement to reset one variable.
2	1.53 $\Rightarrow$ 1.54	Large	Add one variable, and related statements to update this variable when necessary. One statement is updated to use this variable instead.
3	1.54 $\Rightarrow$ 1.55	Medium	Modify the condition of two “if” statements.
4	1.55 $\Rightarrow$ 1.56	Large	Move one function. Add one “if” statement to reset one variable when it’s invalid, and all the other four related variables.
5	1.56 $\Rightarrow$ 1.57	Medium	Move two “memcpy” statements to be next to the relative “if” statements.
6	1.57 $\Rightarrow$ 1.58	Large	Modify the condition of two “if” statements. Add two “for” loops. Remove two statements.

Fig. 22. Case study update details.

greatly. This could significantly reduce the code similarity in the final binary. For example, when we added a global variable in CntToLeds, we observed 517 instruction differences which accounts for about 10% of the total instructions.

In our second test case, D2, we shuffled the global variables in the code and changed their names. Interestingly, no code change was observed in GCC-RA unless the variable names were changed. This is because the data allocation scheme in gcc hashes the variable into the symbol table using their names. This helps to improve the compilation speed, but it also creates difficulties under the update-conscious requirement. For example, a newly added variable may be allocated to the beginning of the data segment, causing many changes in data layout, even if it is defined at the end of a function. Similarly, even for functions with few variables, this is difficult to attack as the function may be inlined during optimizations. Notice that a name change of a variable is essentially a deletion of the old variable plus an insertion of a new variable. This can be handled naturally by UCC-DA as the new variable always takes the space of a deleted variable. Therefore, the change of variable names can be solved easily with UCC-DA to improve the code similarity.

6.8 Case Study

We used function unit Deluge [Hui and Culler 2004] of TinyOS-1.x [TinyOS 2008] as case study. Deluge [Hui and Culler 2004] is a reliable data dissemination protocol for propagating large amount of data over the WSNs. We studied updates of Deluge from TinyOS version 1.52 to version 1.58. The update details are shown in Figure 22.

We used both GCC and UCC to get new binaries after each update and then generate the update scripts accordingly using the update primitives described in Section 5. Figure 23 shows the comparison results, which include the number of script instructions per each type of primitive, and the final script size (bytes) for these two compilation techniques. In the case study, we did not add new instructions such that the performance is the same.

The average script size deduction for all six test cases is 55% compared with GCC. This is because UCC reduces the instructions that need to be updated,

Case #	GCC Script Size (bytes)	#A	#R	#P	#C	#L	UCC Script Size (bytes)	#A	#R	#P	#C	#L
1	249	4	3	22	30	0	5	1	0	0	2	0
2	557	6	3	52	62	0	191	8	3	1	4	0
3	447	2	1	39	43	0	12	3	3	0	4	0
4	605	1	1	4	7	0	512	6	0	1	8	0
5	277	0	4	31	36	0	35	3	1	0	3	0
6	3,981	12	6	143	162	0	1,069	2	2	1	6	2

Fig. 23. Case study script size comparison (#A: add primitive; #R: remove primitive; #P: replace primitive; #C: copy primitive; #L: clone primitive).

Fig. 24. Duration time comparison.

thus the number of update primitives in the script is reduced. In addition, we found that when more instructions need to be updated, the code tends to be divided into smaller pieces, which results in more copy primitives. For example in Case 3, UCC reduces the number of replace primitives from 39 to 0, and copy primitives from 43 to 4, which results in a 97% script-size deduction. From the results, we also observed that it is not always beneficial to apply the clone primitive. When the number of the instructions that can be “cloned” from the original code is not big enough, using replace primitive is more efficient.

Fig. 25. Data transmission comparison.

Besides the script size comparison, we also used the software update protocol Deluge [Hui and Culler 2004] to disseminate the scripts to the WSN and measured the completion time and data size transmitted in the network. We used TOSSIM [Levis et al. 2003], the simulator provided by TINYOS [TinyOS 2008] to disseminate these scripts over a 5×5 network. The scripts are injected to the sink node and then propagated to the whole network.

We first compared the update duration time, which measures the time spent on routing these scripts in the network. It starts when the script injection to the sink node is finished and ends when the whole network receives the scripts. The results are shown in Figure 24. As larger scripts cost more time to propagate in the network and UCC generates smaller scripts comparing with GCC, according to our experiment results, we can save 29% dissemination time, on average.

Then, we compared the transmission overhead over the network between UCC and GCC. We measured the size of the data transmitted over the network, which includes all three types of messages that are used by Deluge [Hui and Culler 2004]. Because the scripts are divided into fixed-sized pages to transmit over the network, larger scripts produce more pages, which results in a high-transmission overhead. In addition, more pages tend to cause more

communication conflicts over the network. On average, UCC transmits 46% less data during code dissemination.

7. RELATED RESEARCH

We have discussed prior research that are closely related to our work in the introduction section. Here, we focus on the previous work on register allocation in a traditional compilation framework.

In the past 20 years, the register allocation problem has been extensively studied with great success in many aspects. Traditional register allocators construct the interference graph of variables and solve the global register allocation as a graph coloring problem [Chaitin et al. 1981; George and Appel 1996; Briggs et al. 1994; Chow et al. 1984]. To achieve fast compilation, linear-scan algorithms assign variables to available registers through a simple scan of the program, instead of constructing the interference graph [Poletto et al. 1999; Traub et al. 1998]. It was reported that linear-scan allocators generate similar code as those from graph coloring-based allocators. Recently, the optimal or near-optimal register allocation was formulated and solved through ILP [Goodwin and Wilken 1996] or multicommodity network flows [Koes et al. 2006]. In addition to achieving better performance, algorithms have been proposed to achieve many other objectives. For example, an early work [Naik and Palsberg 2002] considered the code size constraint in register allocation, and generated compact code for embedded systems. Researchers also [Zhuang and Pande 2006] exploited differential encoding designs that allow the use of more registers in the program.

To save the compilation time after small code changes, Bivens and Soffa [1990] proposed the incremental register allocation (IRA) scheme which, similar to UCC, only reallocates registers for the changed code while preserving the assignment for unchanged code. The difference is IRA exploits the traditional graph-coloring algorithm for the changed code without considering code similarity and energy-consumption model. IRA uses a different criteria to identify changed code and it neither performs interregister movement nor gives register selection priority to preferred registers. The register assignment generated from IRA is not update conscious.

8. CONCLUSIONS

In this article, we proposed UCC techniques for achieving energy efficiency in WSNs. We present algorithms on how to perform update-conscious data allocation and register allocation. The experimental results showed great improvements over update-unconscious solutions. In the future, we will extend the update-conscious compilation research to other environments using costly wireless communication, such as cellular phone users in ad hoc networks.

ACKNOWLEDGMENTS

We thank Dr. Kim Hazelwood and anonymous reviewers for their insightful comments for improving the article. We also thank Dr. Bruce R. Childers and Yu Du for the helpful discussions regarding this work.

REFERENCES

- BARR, K. AND ASANOVIĆ, K. 2006. Energy aware lossless data compression. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*. ACM, New York, 250–291.
- BRIGGS, P., COOPER, K. D., AND TORCZON, L. 1994. Improvements to graph coloring register Allocation. *ACM Trans. Program. Lang. Syst.* 16, 33, 428–455.
- BERKELAAR, M. 2006. LP solve 5.5. <http://cran2.armsmachinandi.it/doc/packages/lpSolve.pdf>
- BIVENS, M. P. AND SOFFA, M. L. 1990. Incremental register allocation. *Softw. Pract. Exp.* 20, 10, 1015–1047.
- CALLAWAY, JR., E. H. 2003. *Wireless Sensor Networks: Architectures and Protocols*. CRC Press, Boca Raton, FL.
- CHAITIN, G. J., AUSLANDER, M. A., CHANDRA, A. K., COCKE, J., HOPKINS, M. E., and MARKSTEIN, P. W. 1981. Register allocation via coloring. *Comput. Lang.* 6, 45–57.
- CHOW, F. AND HENNESSY, J. 1984. Register allocation by priority-based coloring. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*. ACM, New York, 222–232.
- CROSSBOW. HMica2 Wireless Measurement System. <http://www.xbow.com>.
- DAI, W. The Crypto++ Library. <http://www.eskimo.com/weidai/cryptlib.html>
- DIFFUTILS. Home page. <http://www.gnu.org/software/diffutils>
- DUNKELS, A., FINNE, N., ERIKSSON, J., AND VOIGT, T. 2006. Runtime dynamic linking for reprogramming wireless sensor networks. In *Proceedings of the ACM International Conference on Embedded Networked Sensor Systems (SenSys'06)*. ACM, New York, 15–28.
- DUTTA, P. K., HUI, J. W., CHU, D. C., AND CULLER, D. E. 2006. Securing the deluge network programming system. In *Proceedings of the International Symposium on Information Processing in Sensor Networks (IPSN'06)*. ACM, New York, 326–333.
- GOODWIN, D. W. AND WILKEN, K. D. 1996. Optimal and near-optimal global register allocations using 0/1 integer programming. *Softw. Pract. Exp.* 26, 8, 929–965.
- GEORGE, L. AND APPEL, A. W. 1996. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.* 18, 3, 300–324.
- HUI, J. W. AND CULLER, D. E. 2004. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the ACM International Conference on Embedded Networked Sensor Systems (SenSys'04)*. ACM, New York, 81–94.
- JEONG, J. AND CULLER, D. E. 2004. Incremental network programming for wireless sensors. In *Proceedings of the 1st Annual IEEE Sensor and Ad Hoc Communications and Networks (SECON'04)*. IEEE, Los Alamitos, CA, 25–33.
- JUANG, P., OKI, H., WANG, Y., MARTONOSI, M., PEH, L.-S., AND RUBENSTEIN, D. 2002. Energy efficient computing for wildlife tracking: Design tradeoffs and early experiences with ZebraNet. In *Proceedings of the ACM/IEEE International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'02)*. ACM, New York, 96–107.
- KAHN, J. M., KATZ, R. H., AND PISTER, K. S. J. 2000. Emerging challenges: Mobile networking for “Smart Dust.” *J. Comm. Netw.* 2, 3, 188–196.
- KOES, D. R. AND GOLDSTEIN, S. C. 2006. A global progressive register allocator. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06)*. ACM, New York, 204–215.
- KOSHY, J. AND PANDEY, R. 2005. Remote incremental linking for energy-efficient reprogramming of sensor networks. In *Proceedings of the European Workshop on Wireless Sensor Networks*. Springer, Berlin, 354–365.
- KULKARNI, S. S. AND WANG, L. 2005. Mnp: Multihop network reprogramming service for sensor networks. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS'05)*. IEEE, Los Alamitos, CA.
- LANIGAN, P. E., GANDHI, R., AND NARASIMHAN, P. 2006. Sluice: Secure dissemination of code updates in sensor networks. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS'06)*. IEEE, Los Alamitos, CA.
- LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. H. 1997. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the International Symposium on Microarchitecture (MICRO'97)*. IEEE, Los Alamitos, CA, 330–335.

- LEVIS, P. AND CULLER, D. 2002. Mate: A tiny virtual machine for sensor networks. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'02)*. ACM, New York, 85–95.
- LIAO, S., DEVADAS, S., KEUTZER, K., TJANG, S., AND WANG, A. 1995. Storage assignment to decrease code size. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95)*. ACM, New York, 235–253.
- NAIK, M. AND PALSBERG, J. 2002. Compiling with code-size constraints. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'02)*. ACM, New York, 120–129.
- MARRON, P. J., GAUGER, M., LACHENMANN, A., MINDER, D., SAUKH, O., AND ROTHERMEL, K. 2006. FlexCup: A flexible and efficient code update mechanism for sensor networks. In *Proceedings of the European Workshop on Wireless Sensor Networks (EWSN'06)*. Springer, Berlin, 212–227.
- NONLINEAR MIXED INTEGER PROGRAMMING. Welcome to the Bonim Wiki. <http://projects.coin-or.org/Bonmin>
- PANTA, R. K., KHALIL, I., AND BAGCHI, S. 2007. Stream: Low overhead wireless reprogramming for sensor networks. In *Proceedings of the IEEE Conference on Computer Communications (Infocom'07)*. IEEE, Los Alamitos, CA.
- VON PLATEN, C. AND EKER, J. 2006. Feedback linking: Optimizing object code layout for updates. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'06)*. ACM, New York, 2–11.
- POLETTI, M. AND SARKAR, V. 1999. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.* 21, 5, 895–913.
- RELJERS, N. AND LANGENDOEN, K. 2003. Efficient code distribution in wireless sensor networks. In *Proceedings of the International Workshop on Wireless Sensor Network Architecture*. Springer, Berlin, 60–67.
- LEVIS, P., LEE, N., WELSH, M., AND CULLER, D. 2003. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys'03)*. ACM, New York, 126–137.
- SHNAYDER, V., HEMPSTEAD, M., CHEN, R.-R., ALLEN, G. W., AND WELSH, M. 2004. Simulating the power consumption of large-scale sensor network applications. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys'04)*. ACM, New York, 188–200.
- TinyOS. Homepage. <http://www.tinyos.net>
- TITZER, B. L., LEE, D. K., AND PALSBERG, J. 2005. Avrora: Scalable sensor network simulation with precise timing. In *Proceedings of the International Symposium on Information Processing in Sensor Networks (IPSN'05)*. ACM, New York, 477–482.
- TRAUB, O., HOLLOWAY, G., AND SMITH, M. D. 1998. Quality and speed in linear-scan register Allocation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*. ACM, New York, 142–151.
- YE, F., ZHONG, G., LU, S., AND ZHANG, L. 2005. Gradient broadcast: A robust data delivery protocol for large scale sens or networks. *ACM Wirel. Netw.* 11, 2, 285–298.
- ZHUANG, X. AND PANDE, S. 2006. Differential register allocation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06)*. ACM, New York, 168–179.

Received September 2008; accepted June 2009