

**SOFTWARE UPDATE MANAGEMENT IN
WIRELESS SENSOR NETWORKS**

by

Weijia Li

B.S., Nanjing University, 2003

Submitted to the Graduate Faculty of
the Department of Computer Science in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2011

UNIVERSITY OF PITTSBURGH
COMPUTER SCIENCE DEPARTMENT

This dissertation was presented

by

Weijia Li

It was defended on

October 22nd, 2010

and approved by

Dr. Youtao Zhang, Department of Computer Science

Dr. Daniel Mossé, Department of Computer Science

Dr. Bruce Childers, Department of Computer Science

Dr. Daqing He, School of Information Sciences and Intelligent System Program

Dissertation Director: Dr. Youtao Zhang, Department of Computer Science

SOFTWARE UPDATE MANAGEMENT IN WIRELESS SENSOR NETWORKS

Weijia Li, PhD

University of Pittsburgh, 2011

Wireless sensor networks (WSNs) have recently emerged as a promising platform for many non-traditional applications, such as wildfire monitoring and battlefield surveillance. Due to bug fixes, feature enhancements and demand changes, the code running on deployed wireless sensors often needs to be updated, which is done through energy-consuming wireless communication. Since the energy supply of battery-powered sensors is limited, the network lifetime is reduced if more energy is consumed for software update, especially at the early stage of a WSNs life when bug fixes and feature enhancements are frequent, or in WSNs that support multiple applications, and frequently demand a subset of sensors to fetch and run different applications.

In this dissertation, I propose an energy-efficient software update management framework for WSNs. The diff-based software update process can be divided into three phases: new binary generation, diff-patch generation, and patch distribution. I identify the energy-saving opportunities in each phase and develop a set of novel schemes to achieve overall energy efficiency.

In the phase of generating new binary after source code changes, I design an update-conscious compilation approach to improve the code similarity between the new and old binaries. In the phase of generating update patch, I adopt simple primitives in the literature and develop a set of advanced primitives. I then study the energy-efficient patch distribution in WSNs and develop a multicast-based code distribution protocol to effectively disseminate the patch to individual sensors.

In summary, this dissertation successfully addresses an important problem in WSNs. Update-conscious compilation is the first work that compiles the code with the goal of improving code similarity, and proves to be effective. The other components in the proposed framework also advance the state of the art. The proposed software update management framework benefits all WSN users, as software update is indispensable in WSNs. The techniques developed in this framework can also be adapted to other platforms such as the smart phone network.

TABLE OF CONTENTS

ACKNOWLEDGEMENT	xiii
1.0 INTRODUCTION	1
1.1 Overview of this research	5
1.1.1 Contributions	8
1.2 Assumptions	8
1.3 Organization of this dissertation	11
2.0 BACKGROUND AND RELATED WORK	12
2.1 Software update in WSNs	12
2.2 Compiler	13
2.2.1 Register allocation	14
2.2.2 Data allocation	16
2.3 Patch generator	18
2.4 Distribution protocol	19
3.0 UPDATE-CONSCIOUS COMPILER (UCC)	21
3.1 An overview of UCC	21
3.2 UCC techniques for general-purpose applications	25
3.2.1 UCC data allocation (UCC-DA) for general-purpose applications	25
3.2.1.1 Data allocation problem for general-purpose applications	25
3.2.1.2 UCC data allocation for general-purpose applications	26
3.2.2 UCC register allocation (UCC-RA) for general-purpose applications	32
3.2.2.1 Register allocation problem for general-purpose applications	32
3.2.2.2 UCC register allocation for general-purpose applications	33

3.2.3	The integration of UCC-DA and UCC-RA	44
3.2.3.1	Performing UCC-DA and UCC-RA in one step	45
3.2.3.2	Performing UCC-DA and UCC-RA in two steps	47
3.3	UCC techniques for DSP applications	48
3.3.1	Data allocation problem for DSP applications	49
3.3.2	UCC data allocation (UCC-DA) for DSP applications	50
3.3.2.1	Coalescing single offset assignment (CSOA)	51
3.3.2.2	Incremental coalescing single offset assignment (ICSOA)	53
3.3.3	Incremental coalescing general offset assignment (ICGOA)	55
4.0	SOFTWARE DIFFERENTIAL PATCHING	58
4.1	Instruction-based patching	59
4.1.1	Simple primitives	59
4.1.2	Advanced primitives	61
4.1.2.1	The <code>shift</code> primitive	61
4.1.2.2	The <code>clone</code> primitive	63
4.1.2.3	The <code>insert_access</code> primitive	66
4.1.3	Sensor-side primitive interpretation	68
4.2	Data-based patching	71
4.2.1	Data-based primitives	72
4.2.2	Sensor-side primitive interpretation	72
4.2.2.1	Auxiliary data structures	74
5.0	DISTRIBUTION PROTOCOL	78
5.1	Broadcast-based code distribution protocols	78
5.1.1	Deluge: an effective code dissemination protocol for SA-WSNs	78
5.1.2	Melete: a controlled broadcasting protocol for MA-WSNs	80
5.2	MCP: a Multicast-based code redistribution protocol	81
5.2.1	An overview of the protocol	81
5.2.2	ADV message and application information table (AIT)	82
5.2.3	Request multicasting	85
5.2.4	Caching	86

6.0 EXPERIMENTAL RESULTS	88
6.1 Construction of the update benchmark suite	88
6.1.1 Test case categorization	89
6.1.2 real-bench : real test cases	91
6.1.3 man-bench : manually generated test cases	93
6.1.4 auto-bench : automatically generated test cases	93
6.1.4.1 Methodology used to generate the auto-bench	96
6.2 Patch generation and dissemination in SA-WSNs	96
6.2.1 Updating general-purpose applications using UCC-RA	96
6.2.1.1 Settings	97
6.2.1.2 The generated script size	97
6.2.1.3 The generated code quality	100
6.2.1.4 The energy savings	100
6.2.1.5 The problem complexity and compilation time	102
6.2.2 Updating general-purpose software using UCC-DA	105
6.2.2.1 Settings	106
6.2.2.2 The generated script size	106
6.2.2.3 The energy savings	107
6.2.2.4 The wasted memory space	107
6.2.2.5 Tradeoff between wasted space and binary differences	109
6.2.3 Updating general-purpose applications using UCC-RA and UCC-DA	110
6.2.3.1 Performance evaluation using man-bench	110
6.2.3.2 Performance evaluation using real-bench	111
6.2.4 Updating DSP applications	113
6.2.4.1 Settings	113
6.2.4.2 Script size comparison using man-bench	113
6.2.4.3 Code quality comparison using man-bench	115
6.2.4.4 Performance evaluation using auto-bench	119
6.2.4.5 Performance evaluation using real-bench	121
6.3 Software update strategy in SA-WSN	123

6.4 Patch dissemination within MA-WSNs	124
6.4.1 Settings	125
6.4.2 Message overhead	125
6.4.3 Completion time	126
6.4.4 Sensitivity to node distribution	127
6.4.5 Sensitivity to application sizes	128
6.4.6 Sensitivity to cache sizes	128
7.0 FUTURE DIRECTIONS AND CONCLUSION	131
7.1 Future work	131
7.1.1 Apply to different platforms	131
7.1.2 Other update-conscious compilation schemes	132
7.2 Conclusion	133
8.0 BIBLIOGRAPHY	135

LIST OF FIGURES

1	A Mica2 sensor and its block diagram.	1
2	A Imote2 sensor and its block diagram.	2
3	Software upgrade in a WSN.	4
4	Software switch in a MA-WSN.	5
5	An overview of the software update management framework.	6
6	Compiler work flow.	13
7	An example of DSP code generation.	17
8	Basic code distribution protocol (SPIN).	20
9	Sink-side update-conscious compilation.	22
10	Sensor-side code update and execution.	22
11	Placing update conscious compilation (UCC) in the traditional compilation flow.	24
12	An example of incremental data allocation.	26
13	The sensor memory model.	30
14	An example of data allocation for general-purpose applications.	31
15	An example of register allocation for general-purpose applications.	32
16	The decision variables used in UCC-RA.	35
17	The objective function used in UCC-RA.	41
18	The notation used in the UCC-RA objective function.	42
19	The objective function used in ILP-based UCC integration.	46
20	The converted objective function used in the ILP based UCC integration.	47
21	The notation used in the two-step integration.	47
22	The objective function used in the two-step approach.	48

23	An example of data allocation for DSP applications.	50
24	An example of the need for UCC data allocation for DSP applications.	51
25	The update script comparison between CSOA and the update-conscious scheme.	52
26	An overview of the ICSOA-based code update scheme.	53
27	An example of ICSOA scheme.	55
28	Patch generation and binary reconstruction.	58
29	The instruction-based patch script primitives	60
30	An example of the simple primitives.	60
31	An example of the <code>shift</code> primitive.	62
32	An example of the <code>clone</code> primitive.	64
33	An example of the <code>insert_access</code> primitive.	66
34	An example of the interpretation procedure of the <code>insert_access</code> primitive.	67
35	The primitives to patch the data allocation.	72
36	The code construction procedure of the data-based primitives.	73
37	Coalesced variable list.	74
38	The AR in/out value list.	75
39	Advertise-request-data handshaking protocol in Deluge.	79
40	An example of software switch in a multi-application WSN (MA-WSN).	80
41	An example of the application information table (AIT).	83
42	Gradient-based request routing.	86
43	Base programs for the construction of the software update benchmark.	89
44	Real general-purpose application update benchmark.	92
45	Real DSP application update benchmark.	92
46	Manually generated general-purpose application update benchmark.	94
47	Manually generated DSP application update benchmark.	95
48	Script size comparison between UCC-RA and GCC-RA.	98
49	Code quality comparison between UCC-RA and GCC-RA.	99
50	The energy savings for general-purpose applications.	101
51	The compilation time of UCC-RA.	102
52	The number of constraints as a function of the number of IR instructions.	103

53	The number of iterations as a function of.	104
54	The time to solve one iteration as a function of.	105
55	Script size comparison between UCC-DA and GCC-DA.	107
56	Energy savings using UCC-DA.	108
57	Worst-case stack size comparison between UCC-DA and GCC-DA.	108
58	Tradeoff between the worst-case stack size and the instruction updates.	109
59	Script size comparison between the integrated scheme and the baseline scheme.	111
60	Script size comparison for real purpose updates.	112
61	Script size comparison between ICSOA and CSOA (#addr register=1).	114
62	Script size comparison ICGOA and CGOA (#addr register=2).	114
63	Code quality comparison between CSOA and ICSOA.	116
64	Execution overhead breakdown.	116
65	Code quality comparison between ICGOA and CGOA.	118
66	The energy savings for DSP applications.	118
67	Script size comparison using <code>auto-bench</code>	120
68	Code quality comparison using <code>auto-bench</code>	121
69	Script size comparison between CSOA and ICSOA (#addr register=1).	122
70	Script size comparison between CGOA and ICGOA (#addr register=2).	123
71	Code quality comparison between CSOA and ICSOA (#addr register=1).	124
72	Message overhead.	125
73	Dissemination time.	126
74	Dissemination with different numbers of sources and requesters.	127
75	Dissemination with uneven source/requester node distribution.	128
76	Dissemination with different number of pages.	129
77	Dissemination with Different Cache Sizes.	129

LIST OF ALGORITHMS

1	UCC-DA for general-purpose applications.	28
2	Incremental coalescing-based SOA (ICSOA)	54
3	Incremental coalescing-based GOA (ICGOA).	56
4	Primitive interpretation and code reconstruction.	69
5	write_code_buffer /*write the constructed code into code buffer*/	71
6	addr_mode_correction /*Correct the addressing mode of an instruction*/	77

ACKNOWLEDGEMENT

First, I would like to express my deepest gratitude to my advisor Dr. Youtao Zhang, for his invaluable guidance, for keeping me on track and pushing me in my research, for all his practical career advice, and for generously supporting me as a graduate student. It was my great fortune to have found such a wonderful adviser.

Second, I would like to thank my committee members Dr. Daniel Mossé, Dr. Bruce Childers and Dr. Daqing He for their suggestions and feedback on my dissertation.

Also, I want to thank my friends, particularly Dr. Jonathan Derryberry for giving me advice and encouragement along the way as I navigated through the graduate program.

Additionally, I want to thank Jonathan for encouraging me to finish my dissertation and providing me occasional refreshing distractions.

Last but not least, I would like to dedicate this thesis to my parents. Without their support, I can hardly make it done. I would also like to thank my grandfather, whom I miss dearly. I wish he could have been here to see me getting my doctor's degree.

1.0 INTRODUCTION

Wireless sensor networks (WSNs) [12, 36, 37] have recently emerged as a promising platform for many non-traditional applications, such as wildfire monitoring and battlefield surveillance. A WSN usually consists of tens to hundreds of sensors that can sense physical phenomena, such as temperature, humidity, pressure, and movement of objects. Sensing results are optionally preprocessed, split into data packets, and then routed back to the sink nodes that are more powerful and user-accessible, and have no or negligible energy constraints.

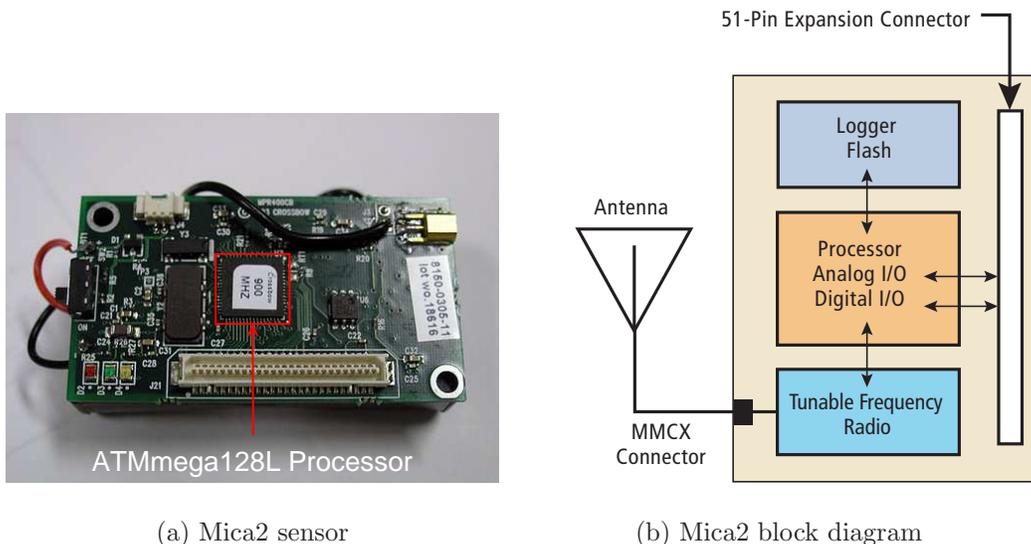


Figure 1: A Mica2 sensor and its block diagram [19].

A sensor node typically contains sensing devices for data collection, communication transceivers for sending and receiving data packets, and processing units for arithmetic and logic computation. Powered by one or two batteries, most deployed sensors have only a limited energy supply, which makes energy saving one of the most important design criteria in

WSNs. A common approach for saving energy is to choose a low-power processor [19, 64]. For example, a Mica2 sensor (shown in Figure 1) uses an 8MHz ATmega128L micro-controller to process the sensed data.

Since transmitting one bit one hop in a WSN consumes about the same energy to execute 1000 instructions [5], another approach for saving energy is to pre-process the collected data and forward only the data summary to the sink node, which saves transmission energy due to generating less traffic in the network. Pre-processing some types of data, e.g., multimedia data, is often too energy-expensive for low-power processors. Therefore, a sensor may add coprocessors to achieve energy efficiency. For example, the Imote2 [19] node developed by Intel (shown in Figure 2) includes an *extra* digital signal processor (DSP) coprocessor to process the audio and video data collected by its associated microphone and camera. Nachman *et al.* showed that the DSP coprocessor in Imote2 implements a WMMX (wireless multimedia instruction set extension) instruction set and can achieve up to 16× speedup when implementing filter kernels [57].

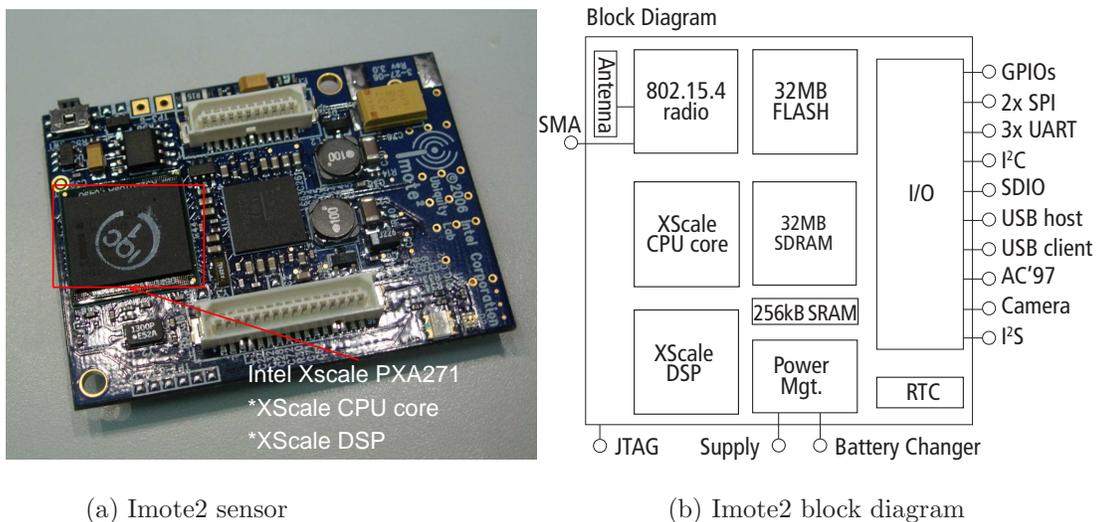


Figure 2: A Imote2 sensor and its block diagram [19].

The availability of multi-core sensors exposes more design opportunities. However, the high manufacturing cost of these sensor nodes makes it economically less appealing to let the whole network run just one application. Recently, researchers have envisioned the wide

adoption of multi-application wireless sensor networks (MA-WSNs), which support executing multiple applications in one network infrastructure [7, 74, 83].

Compared to single-application wireless sensor networks (SA-WSNs), MA-WSNs have many advantages in efficiency and flexibility. For example, a MA-WSN can be deployed in a national park to monitor both wildfires and animal movements. A greater proportion of sensors can be set to monitor animal movement during seasonal migration, and more sensors can be set to monitor wildfires during the summer when wildfires are more likely to occur. By using the same network infrastructure for both applications, MA-WSNs can achieve two goals. First, they amortize the investment needed to deploy multiple sensor networks in the same area; and second, they can adapt to the changing environment and adjust the coverage on demand.

In both SA-WSNs and MA-WSNs, the software may need to be updated for various reasons. In this dissertation, I separate these reasons into two categories: software upgrade and software switch. *Software upgrade* refers to the problem of updating one application with a newer version for all nodes in a WSN. In contrast, *software switch* refers to the problem of updating a subset of nodes with an existing application in a WSN.

Software upgrade. After deploying a WSN, the applications running on the sensor nodes may need to be upgraded. Bug fixes and feature enhancements are common reasons for software upgrade. Updates are particularly frequent when applications are still in the development stage, as testing and debugging may take several rounds until the code is stable. For example, a WSN may be deployed in an unfamiliar area so that the preliminary data can help scientists better calibrate their sensing applications. As shown in Figure 3, software upgrade involves binary code generation on the sink node, patch generation, patch distribution, and sensor-side binary replacement. Because sensors are usually left unattended after deployment, the patch deployment can only be done via wireless communication, an expensive operation in WSNs. A recent study [5] showed that the energy consumed to send one bit one hop in a WSN is about the same as the energy consumed to execute 1000 instructions. For large WSNs in which the sink node cannot reach all sensors via broadcasting, the patch has to be transmitted hop-by-hop in the network, which consumes a significant amount of the energy of each sensor node. As sensor nodes are running with

a limited energy supply, it is essential to conserve the energy in a WSN during software updates, especially when such updates are frequent.

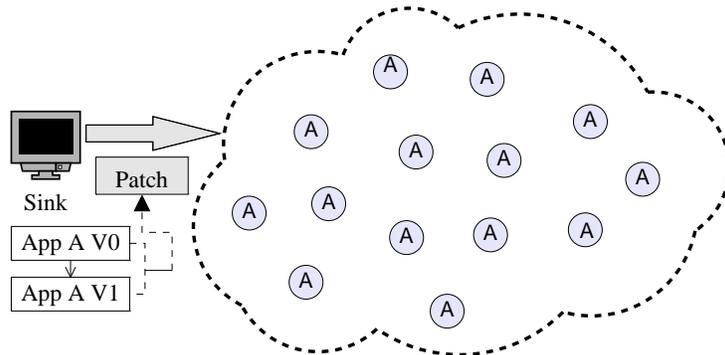


Figure 3: Software upgrade in a WSN.

One possible solution for saving energy is to reduce the number of transmitted bytes during software upgrade, which saves transmission energy. This can be attained through three consecutive and cooperative steps — (1) Since the update patch represents the binary difference of the new and old code images in my setting (Section 1.2 discusses the tradeoffs of alternative code formats), it is important to minimize the binary level code difference; (2) Given the new and old code images, a good patch generator can help generate a minimized patch; (3) Given a generated patch, a good code distribution protocol can further minimize the network traffic in the dissemination process.

Software switch. Altering the code images of sensors in a MA-WSN is more complicated. To support running multiple applications in a MA-WSN, it is possible to preload multiple code images to the sensor nodes before deployment, and switch among them upon request from the sink node. However, due to the memory size constraint, not all code images can be stored on each sensor. This indicates that some sensor nodes need to fetch the binary of the wanted yet unavailable application from a source node. The source node can be either the sink node, or the neighboring sensors that own the requested code image. Figure 4 shows a software switch example in which only a subset of sensors running application B or C need to switch to A. Software switch differs from software upgrade in that: (i) only a subset of sensors need to be updated in software switch while all nodes need to be updated in software upgrade; and (ii) both the sink node and many remote sensors can act as source nodes in

software switch while only the sink node is the source node in software upgrade.

In MA-WSNs, many different applications may share the same or similar functionalities and thus the identical code pieces, e.g., the code for sending and receiving messages. Based on this observation, when switching one application to the other, a possible solution for saving energy is to generate the same binary for the shared code, and transmit only the difference between two applications rather than the entire binary of the other application. Software switch, similar to software upgrade, also requires an effective patch generator and code distribution protocol to achieve energy efficiency.

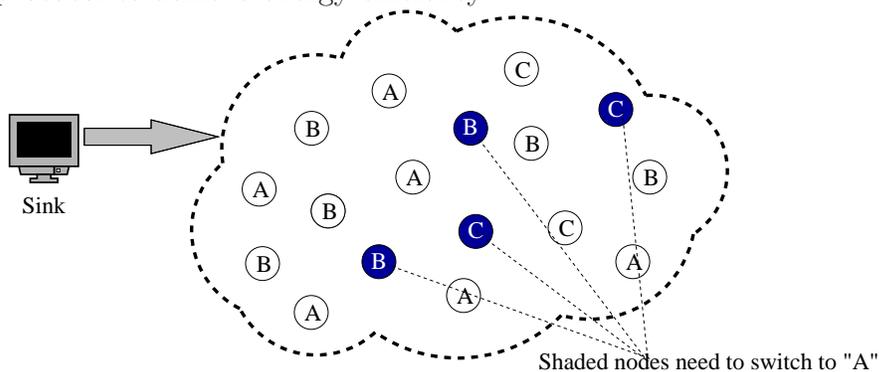


Figure 4: Software switch in a MA-WSN.

To summarize, software updates (either software upgrades or software switches) can occur frequently in WSNs. Relying heavily on costly wireless communication, such updates could consume a significant amount of energy and greatly shorten the network lifetime. In addition, when such updates are in progress, the WSN is usually in the *down* mode, i.e., it cannot provide the designed service [61]. Therefore, how to design an efficient framework that minimizes both the energy consumption and the downtime during the software update is an important problem to study in WSN research.

1.1 OVERVIEW OF THIS RESEARCH

In this section, I present an overview of my proposed research. It consists of the following steps in the software update procedure. First, the compiler generates the binary image(s).

Second, the patch generator produces the patch. Third, the patch is distributed to the WSN. After receiving the complete patch, each sensor regenerates the target binary, loads it to memory, and starts running it.

Figure 5 illustrates my proposed software update management framework. The sink node is a computer server that has no resource constraints, while the sensors have tight resource limitations in energy, memory size, network bandwidth, and computation ability. To update the code on sensors, the compiler first translates the source code S' into an executable binary E' . Instead of sending E' , a small patch P that contains only the difference between E' and the original binary E is distributed. When the code distribution is complete, each sensor regenerates E' by combining the received patch P and the preloaded binary E .

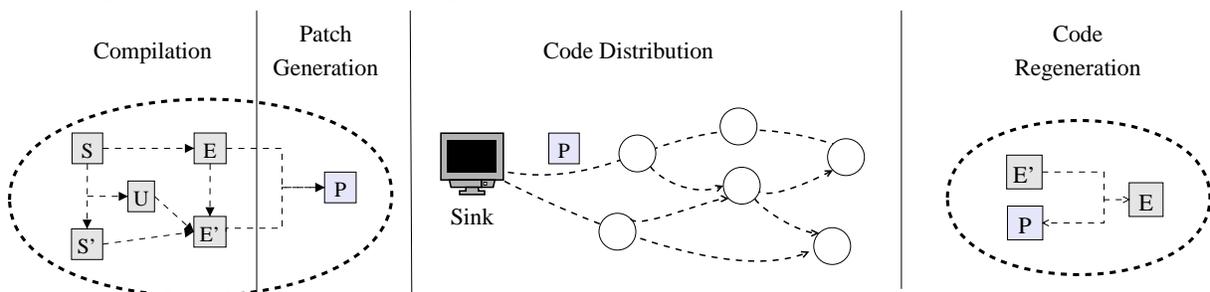


Figure 5: An overview of the software update management framework.

This framework includes three major components.

Update-conscious compiler. Since the patch transmitted over the network is the difference between the old binary and the new binary, increasing the binary similarity between the two versions can reduce the number of bytes that need to be transmitted, resulting in saving both energy consumption and transmission time in software update.

In most cases, only a subset of source code level statements are updated in the new version. They are referred to as *changed statements* while others are referred to as *unchanged statements*. The binary level differences produced from changed statements are difficult to avoid. However, unchanged statements may also produce binary differences, which are considered unnecessary. This is because an update-unaware compiler randomly picks up a choice if there are multiple alternatives that generate the same code performance (or other criteria) during compilation.

In this dissertation, I will propose update-conscious compiler (UCC) techniques that read the old source and binary to know the compilation choices of the old version, and use them as hints when generating the new binary. As shown in Figure 5, UCC takes E (the old binary), U (the intermediate level differences between the old version and the new version), and S (the new source code) as inputs to generate the new binary E' . By taking the old binary into consideration, my update-conscious compilation improves the code similarity between two code images. This dissertation develops the UCC register allocation and data allocation schemes for different applications. In my future work, additional UCC schemes will be developed in the framework.

When minimizing the binary level difference, UCC may not generate the code that runs as efficiently as the code generated by a conventional compiler. Trading run-time performance for binary similarity aims to save energy during software update and avoid wasting execution energy at runtime. Clearly, a naive design that always maximizes the code similarity may not reduce the total energy consumption. To solve this problem, UCC studies the trade-off between binary code difference and run-time performance, and strives to minimize total energy consumption.

Code distribution protocol. The code distribution protocol disseminates the patch to the sensors that need the update. The code can come from either the sink node or other sensors that own the requested binary image. This dissertation presents a code distribution protocol that achieves energy efficiency under tight resource constraints and works for both software upgrade and software switch.

The protocol must be robust. Since both the wireless links and the nodes may be lost temporarily or permanently in WSNs, the protocol needs to tolerate both link and node failures. The protocol must also be fast in order to minimize the *down* time of the network, i.e., the duration of software update.

Patch generator. The new binary E' needs to be compared with the old binary E to generate the binary-level differences in a highly condensed script P . Multiple binary-level differences may have the same cause; for example, after inserting one instruction in the code, the destination addresses of several branch instructions may change the same constant. This has been discovered in the literature [35, 61, 68]. Including only the root cause instead of

individual changes can effectively reduce the patch size. However, if the update script design is complicated, it requires much effort at the sensor side to decode and regenerate the new binary. This dissertation presents several sets of script primitives, and evaluates the trade-off between patch transmission and sensor-side decoding complexity.

1.1.1 Contributions

To summarize, my contributions to the field of software update management for WSNs are as follows.

- I propose update-conscious compilation (UCC), the first work that takes a compilation approach to improve code similarity for energy-efficient software update. UCC advances the state of the art and uses overall energy efficiency as the metric to make compilation decisions.
- I compose a framework that covers the three major phases of software update in WSNs: binary generation, patch generation, and code distribution. I successfully identify research problems in each phase and propose novel designs that enable the attainment of overall energy efficiency.
- I evaluate the proposed techniques in an integrated framework. The test cases of my in-house benchmark include real-world test cases, manually generated test cases, and automatically generated test cases. The evaluation is much more thorough and complete compared to similar works in the literature.

1.2 ASSUMPTIONS

To simplify the implementation of the software update framework in WSNs, the following assumptions are made.

The binary is transmitted during software update. In this dissertation, I assume that the code disseminated in the network is binary code. An alternative choice is to release source code and perform sensor-side compilation with a pre-installed compiler. Technology

advances support this approach with large memory installed on recently released sensors, e.g., Imote2 has 32MB SDRAM [19]. Other designs install a lightweight virtual machine on sensor nodes, which enables the execution of Java bytecode-like high-level instructions [22, 39, 47]. Releasing applications in source code or bytecode can effectively reduce the update size. However, for the following reasons, I will only consider software update in binary format and leave the problem of updating software in other formats as a future research topic to explore.

- Modern compiler systems are becoming increasingly complicated. A full-fledged compiler includes not only the compiler itself, but also a set of tools and libraries; e.g., a full installation of GCC 2.95 needs more than 100MB disk space [27]. Enabling sensor-side compilation requires having a number of libraries pre-installed on sensors. Since only a small subset of functions in these libraries are actually linked into the binary, pre-installing libraries on sensors is an inefficient way to use precious sensor storage.
- Running a lightweight GCC compiler tends to generate sub-optimal code that wastes execution energy. Instead, the popular cross-compilation approach adopted by TinyOS [76] and other systems uses a full-fledged compiler at the server side to generate fully optimized code for sensor architectures. Running virtual machines also has the execution overhead problem. The energy wasted by executing more instructions can add up to a non-negligible amount in the long run, as I will discuss in the experiments.
- Releasing source code or high-level bytecode also creates privacy and security concerns. For example, because Java bytecode is more vulnerable to reverse engineering [72], an attacker can capture a sensor and spend much less effort to extract the intellectual property from Java bytecode than from a binary image.

The mapping between the source code and the binary can be created. Many compiler optimization techniques may add, delete, and re-order instructions to achieve better run-time performance. Adopting update-conscious compilation before these techniques may reduce code similarity and thus diminish the benefits gained from update-conscious compilation. In this dissertation, I place update-conscious compilation as a separate pass after other compiler optimizations, and leave the optimization pass ordering problem [40, 81] as a future topic to study. We assume that the compiler can create mappings between the optimized

intermediate representation (IR) and the source code-level statements, as discussed in [34]. The mapping helps to identify the changed and unchanged parts at the lower level, and allow update-conscious compilation focusing only on the changed parts. A conservative strategy is adopted in this dissertation, i.e., if a mapping cannot be created for a code segment, my update-conscious compiler considers that code segment as a changed segment.

The characteristics of applications running in the network are known. Based on the functionalities of these applications, and the expert knowledge from application designers, I assume that the number of executions that one application is expected to perform before retirement is known. This information assists in the estimation of execution energy consumption in my framework. For a MA-WSN, I assume that how often one sensor needs to fetch an application from other sensors in the network is also known. This information helps determine the trade-off between consumption of execution energy and transmission energy.

The types of sensors used in the WSN are known. Different sensors have different hardware constraints, i.e., computation abilities and memory limitations. The computation constraint often restricts a remote sensor from running too-complex programs, otherwise the sensor may suffer from large execution energy consumption and slow response to environmental signals. The memory size information helps us determine the number of applications that could fit into one sensor's memory, and also the amount of free memory that can be used to optimize the packet routing.

Multiple software updates on one sensor do not interleave with each other. Although multiple applications can co-exist on a sensor node, I assume only one application may need to update at a time. The works in the literature [61] and my preliminary data show that concurrently updating multiple applications is not beneficial due to signal collision and memory space competition. In practice, multiple applications can be updated sequentially, controlled by the sink node.

1.3 ORGANIZATION OF THIS DISSERTATION

The remainder of this dissertation is organized as follows. Chapter 2 presents the background on software update in WSNs and discusses the works that are related to the three components of my proposed framework, including traditional register allocation, data allocation design, WSN software update script primitive design, and WSN code dissemination protocol design. Chapter 3 elaborates the proposed UCC design and develops UCC register allocation and data allocation schemes for different types of applications. Chapter 4 presents the script primitives that are used to summarize the binary level differences in update patch. In Chapter 5, the patch distribution protocol is presented. Chapter 6 presents and analyzes the experimental results. Chapter 7 addresses the directions for future research and concludes the dissertation.

2.0 BACKGROUND AND RELATED WORK

In this chapter, I will first introduce different software update designs in WSNs, and then discuss the works that are related to the major components of my framework.

2.1 SOFTWARE UPDATE IN WSNS

The software update designs for WSNs can be categorized according to *what* to disseminate. A naive approach is to transmit the complete new binary image to replace the old one on sensors. For example, Deluge [32], the default code distribution scheme in TinyOS [76], takes this approach. The schemes in this category focus mainly on how to split the new image into packets and route these packets to the sensors under the WSN constraints.

Since the new binary and the old binary often share common code segments, transmitting the complete image is not only unnecessary but also a waste of energy. A better approach is the *diff*-based design, which compares the code of successive versions and generates an edit script that summarizes the differences; only the script is transmitted to the remote sensors, where the new code is re-generated by combining 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 sensors, the *diff*-based approach significantly improves energy efficiency and has become popular in WSNs [22, 35, 39, 52, 61, 68]. The major focus in this category is determining how to compare the two binary versions and generate a small edit script. If the new binary is generated using a conventional compiler, a simple change in the source code may result in many changes in the final binary. This has limited the *diff*-based approach to update only small changes such as bug fixes [68].

The third approach is to transmit the code as high-level instructions instead of binaries. By installing a lightweight virtual machine [47], the application code can be represented using virtual machine instructions. Similarly, a dynamic linker [22, 39] can enable the distribution of pre-linked object modules rather than the raw binary. The tradeoff between distributing code at the binary level and at high levels is between runtime overhead and privacy concerns, as discussed in Section 1.2.

The need to update the code after releasing the software exists in many computing environments, such as desktop computing. Distributing the code difference rather than the complete code image is widely adopted in the industry. For example, the Microsoft Systems Management Server (SMS) [54] from Microsoft, the Tivoli configuration manager [33] from IBM, and the OpenView Change and Configuration Management tool [31] from HP distribute only the difference from the server and patch the system automatically. The challenges for designing diff-based software update schemes for WSNs come from the tight resource constraints of WSNs.

2.2 COMPILER

A compiler is a translator that translates a source program written in one language – the *source* language, to its equivalent target code in another language – the *target* language [1]. The work flow of a compiler is shown as Figure 6.

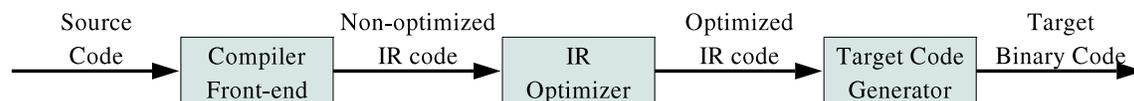


Figure 6: Compiler work flow.

The *front-end* analyzes the source code to build an internal representation of the program, called the *intermediate representation* or *IR*. The *IR* then is transformed into functionally equivalent but faster (or smaller) forms in the *optimizer*. At the last stage, *code generation*, the *optimized IR code* is transformed into the target machine binary.

This dissertation studies two key problems in code generation, *register allocation* and *data allocation*.

2.2.1 Register allocation

Register allocation refers to the problem of deciding *which value* should be held by *which register* at *each program point*. Since there are fewer registers than the values to be held, the values that are not stored in registers reside in the lower-level memory hierarchy, i.e., caches and main memory. Accessing registers is much faster than accessing the values stored in the lower memory hierarchy. Thus, the design goal of traditional register allocation is to find a good allocation strategy to improve program performance. While finding the optimal assignment is mathematically NP-complete, researchers have extensively studied the problem in the past 20 years and have achieved great success in many aspects.

Traditional register allocation schemes. Graph coloring algorithms construct the variable interference graph and solve the global register allocation as a graph coloring problem [11, 13, 16, 28]. To achieve fast compilation, linear-scan algorithms assign variables to available registers through a simple scan of the program, saving the time and space to construct the interference graph [65, 78]. It was reported that the code generated from linear-scan register allocators is only slightly worse than that from graph coloring-based allocators. Register allocation can also be formulated as an integer linear programming (ILP) problem [2, 26, 29] or multi-commodity network flow (MNF) problem [38]. While ILP and MNF enable us to find the optimal or near-optimal allocation results, they are slow and are rarely adopted in commercial compilers.

The register allocation schemes discussed above focus mainly on generating code of better performance in terms of fewer register spills, i.e., memory loads and stores. However, to minimize transmission energy in WSNs, we prefer a register allocator that can improve the code similarity between two versions. Traditional register allocators do not fit the need, as they do not consider code similarity during allocation.

Incremental register allocation. Bivens and Soffa proposed the incremental register allocation (IRA) scheme based on graph coloring [9]. When the software is slightly modified,

this scheme only re-allocates registers for the changed code, and preserves the assignment for the unchanged code.

The goal of IRA is to save compilation time, which is different from my goal of improving code similarity. IRA may generate register allocation results similar to the previous version, yet it always follows the original register allocation for the unchanged code, which may reduce code performance when the source code update is relatively large. As I will discuss and evaluate in later chapters, consuming more execution energy is not always a good choice, even if it can improve code similarity and reduce transmission energy during code update.

Code compression-oriented register allocation. Ros and Sutton proposed a post-compilation register reassignment technique [71]. It creates the mappings of the registers that are used in isomorphic instructions, and tries to replace one register with its mapping register. By increasing the code similarity of different components within *one program*, the scheme helps to improve the compression ratio of Hamming-distance-based code compression [70].

This design does not fit the need, either. The code similarity in my framework refers to that between two code images. In particular, the old image already exists on sensor nodes, and has no need to compress. Improving the code similarity between two images does not mean that these two images need be combined and compressed to a minimized small size.

Prior art and my work. I have discussed the prior art and the reason why existing schemes do not fit the need. To achieve energy-efficient software update in WSNs, this dissertation proposes update-conscious register allocation (UCC-RA). By improving the register allocation similarity of two code images, UCC-RA considers both the transmission energy and the execution energy, and strives to minimize the overall energy consumption. Since only the difference needs to be transmitted, whether the code itself can be highly compressed or not is not important. Since my goal is to save overall energy, generating code of slightly worse performance is acceptable. But for frequently executed code such as loops, maximizing the code similarity with degraded performance is not always a good choice.

2.2.2 Data allocation

Data allocation refers to the problem of assigning variables in a program to memory locations. A traditional compiler usually assigns memory slots to the variables according to the declaration order of these variables. Trishul *et al.* [14] proposed *structure splitting* and *field reordering* to improve cache behavior by reordering the field variables to increase reference locality. Zhang and Gupta [85] proposed compressing field variables of dynamically allocated data structures to improve locality. Data allocation for low-end to middle-end DSP processors has a large impact on the code size and performance due to the unique addressing mode and short instruction width of these processors [6, 51]. I will elaborate further as follows.

Addressing code generation in DSPs. Modern multi-core wireless sensors have integrated DSP co-processors to support processing kernels of multimedia and security applications that handle audio, video, and communication signals [19]. DSP processors can achieve low-cost, low-power, and low-latency digital signal processing by integrating specially optimized architectural components. For example, a dedicated address generation unit (AGU) can perform parallel address computation in *register-indirect* addressing mode. With *register-indirect* addressing, the memory address is stored in an address register (AR) whose value can be automatically updated within a small range before or after memory accesses. Such update-to-address registers incur no extra cost. As a comparison, *base-register-plus-offset* addressing requires two instruction words on 16-bit DSP processors, e.g., AT&T DSP16xx [42].

Because the AGUs on DSP processors assist the address computation in parallel, by carefully allocating variables in the memory, DSP compilers can generate efficient code with compact size and improved performance. For the most frequently used auto-addressing instructions such as post- and pre-address increment/decrement instructions, no explicit addressing instruction is needed when the address distance of two consecutive memory accesses is smaller than 2; and an extra instruction is otherwise needed to update the address register.

The example shown in Figure 7 shows how the data allocation result affects the code generation. Without using auto-addressing instructions, we need two instructions (instruction 20 and 30) to load variable A to R1 and then make the address register AR pointing

to variable B. However, using the post-increment addressing instruction, the two operations can happen in parallel (instruction 20'). By allocating variable A and B next to each other, auto-addressing saves one instruction.

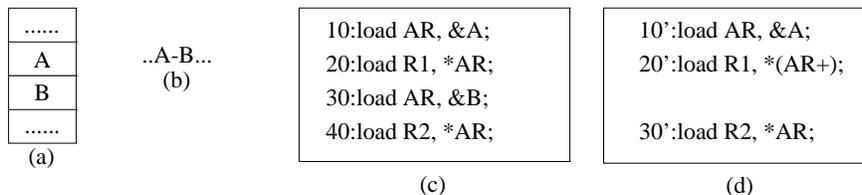


Figure 7: An example of DSP code generation: (a) memory layout; (b) access sequence of the variables; (c) generated instructions without auto addressing; (d) generated instructions with auto addressing.

Offset assignment. From the example, we can see that auto-addressing mode helps to increase the code performance and reduce the code size. Of course, the variables need to be properly allocated in the memory in order to gain these benefits. This problem was first formulated by Bartley [6] and Liao *et al.* [51] as the simple offset assignment (SOA) problem (when there is only one AR), and the general offset assignment (GOA) problem (when there are multiple ARs). A variety of heuristic algorithms have been proposed in the literature [4, 15, 45, 46, 59, 66, 75, 86, 87].

The solution of SOA is equivalent to finding the maximum weight Hamiltonian path¹ in the access graph [6, 51]. The access graph is a graph in which each vertex represents a variable; each edge between two vertices represents that there is at least one consecutive access of two corresponding variables; and the weight of each edge shows the count of such consecutive accesses. Solving a GOA problem is simplified as solving N SOA problems, where N is the number of address registers [51].

Offset assignment with variable coalescing. In DSP applications, many variables have short live ranges. By allocating variables that do not interfere with each other in the same memory location, it is possible to further reduce data memory size and improve code performance. This observation led to variable coalescing heuristics for offset assignment,

¹A Hamiltonian path in a graph is a path that visits every vertex exactly once.

proposed by Ottoni *et al.* [59] and Zhuang *et al.* [86, 87] independently.

Prior art and my work. The design goal of existing offset assignment schemes is to generate code with compact code size and improved performance. When the program is slightly updated, the compiler might generate a different coalesced offset assignment compared to the original version. Even though the memory layout difference is very simple, e.g., when there is a simple switch of two variables’ memory addresses, all the instructions that access these two variables, or the instructions that are adjacent to the memory access instructions of these two variables, may need to apply a different addressing mode, which produces many code differences from the original version.

Instead, I developed an update-conscious data allocation scheme for updates in WSNs that uses the data allocation result from the old version as a hint, while generating the data allocation for the new version. In this way, I can reduce the difference between two versions. I consider both the code performance and the code difference in order to achieve overall energy efficiency.

2.3 PATCH GENERATOR

To support diff-based software update, a patch that summarizes the differences between the new and old code has to be generated.

Binary code patch. The simplest approach to generate a patch is to treat two code images as bit streams and summarize the bit-stream difference. Jeong and Culler [35] proposed to divide the code image into blocks and perform an improved version of brute force search to identify the shared code and the difference. Reijers *et al.* [68] proposed to use a tool that is similar to the `diff` UNIX command, to find the binary code difference. When generating the patch, they introduced address shifting, padding, and address patching primitives to reduce the patch size.

High-level instruction. Patching code at high semantic levels tends to generate a smaller update script. Levis *et al.* showed that code size is very small when using virtual machine instructions [47]. Marrón *et al.* proposed a scheme to produce separate object

files for TinyOS [76] components and let the sensor combine a subset of them together in an executable binary [52]. Dunkels *et al.* further proposed a dynamic linker for this system [22]. Koshy *et al.* proposed to generate relocatable modules and generate the binary using a remote linker [39].

Compression. Compression algorithms, such as bzip2, compress, LZO, PPMd and zlib, can be adopted for software update. Applying compression algorithms on the binary code can reduce the size by 20%~70% [24], which is usually less effective than patching [35, 68]. However, compression can be performed on top of patches to further reduce their sizes. The disadvantage of compression is that a decompressor needs to be installed on the remote sensor. Compression is orthogonal to patch generation and can be smoothly adopted in my framework. I will not discuss and evaluate compression in this dissertation.

Prior art and my work. My work extends existing binary code patching designs [68]. In addition to the simple primitives that have been proposed, I designed and evaluated many advanced and context-aware primitives to further reduce the patch size.

2.4 DISTRIBUTION PROTOCOL

After the patch generation phase, the patches are ready to be distributed over the network. Many code distribution protocols [30, 32, 49, 80, 83] have been proposed.

SA-WSN code distribution protocol. In SA-WSNs, all the sensors run the same application, so the distribution protocol design in SA-WSNs focuses on the efficient flooding scheme which sends the patch packets to all the sensors in the network. The original scheme in WSN, called SPIN [30], uses a three-way handshaking protocol, shown in Figure 8. Each sensor broadcasts the software information (ADV messages) as advertisements. The sensors that need to update their software send request (REQ) messages to the code owners, and then the code owners respond with the data messages.

Trickle [49] improves SPIN by introducing a suppression mechanism to remove unnecessary advertisement messages, which reduces the energy used in the advertisement phase. Deluge [32] extends Trickle to support efficient flooding of large data, especially code images.

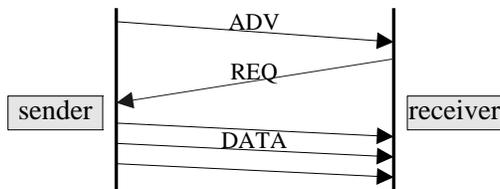


Figure 8: Basic code distribution protocol (SPIN).

It divides a big code image into pages, with each page consisting of multiple packets; packets within a page can be received out of order.

These protocols only support the code update in SA-WSNs — all the sensors get the application updated eventually. For MA-WSNs that have multiple applications, only a subset of all sensors may be running the application to be updated. The distance between the requester and the source can be multiple hops away from each other. The above protocols are not suitable for MA-WSNs, as their messages (advertisement, request, data) are always sent and received in one hop.

MA-WSN code distribution protocol. Melete [83] was proposed to solve the code distribution problem in MA-WSNs. It uses a controlled broadcast strategy to flood sensors within a range. If a source is found, the application can be fetched from a local source. The weakness of this scheme is that it is a stateless protocol — a sensor does not store the routing to the source but relies on the REQ message to discover the routing. This causes single collision and message retransmission, which wastes transmission energy.

Prior art and my work. This dissertation proposes a multicast-based code distribution protocol (MCP). MCP is a stateful protocol — it stores the routing information to the nearby source nodes of each application. Instead of using broadcast, MCP sends out multicast messages, which helps to reduce network traffic and the update finish time.

3.0 UPDATE-CONSCIOUS COMPILER (UCC)

In this chapter, I will first present an overview of the update conscious compilation (UCC) design and then develop UCC data allocation (UCC-DA) and UCC register allocation (UCC-RA) schemes.

This dissertation studies the applications running on two types of architectures that appear in WSNs (as discussed in Chapter 1). The first type of applications, referred to as *general-purpose applications*, are those to be executed on the main sensor processor, e.g., the ATmega128L processor in Mica2 or the XScale core in Imote2 [19]. The code needs to handle interrupts and different types of events, and has a relatively complex control structure. The developed schemes for this type are not architecture-dependent and thus can be adapted to different sensor processors. The second type of applications, referred to as *DSP applications*, are those to be executed on a DSP co-processor, e.g., the DSP core in Imote2 [19]. The code is often the kernel of different filters and security algorithms. The developed schemes for this type exploit the special addressing mode that exists on this type of processors.

3.1 AN OVERVIEW OF UCC

An overview of update-conscious compilation is illustrated in Figure 9. Conventional compilation takes the following steps to generate binary code from the source code. 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, etc.

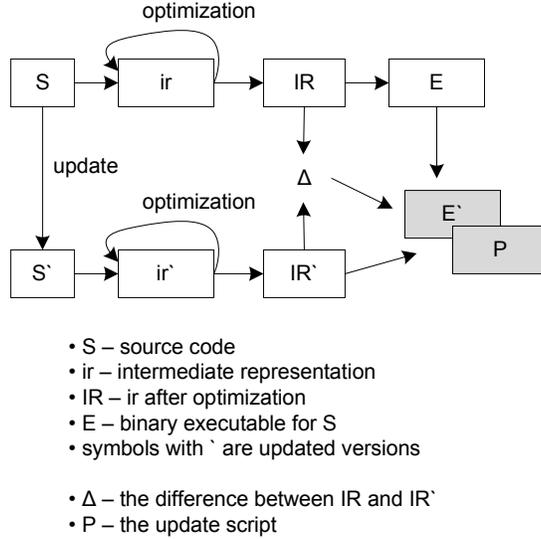


Figure 9: Sink-side update-conscious compilation.

The proposed UCC schemes are performed at the code generation stage, i.e., from IR to E. This helps preserve the performance improvements from the optimization passes. In this dissertation, I will develop three update-conscious schemes for register allocation and data allocation in the code generation stage. For clarity, I assume that the optimization passes are *independent* from these two phases. Other optimization passes will be investigated in future work.

When S is updated to S' (Figure 9), 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 P, is small.

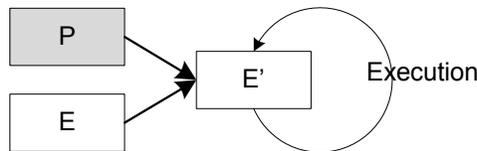


Figure 10: Sensor-side code update and execution.

The binary difference P will then be transmitted over the network to the sensors. When the sensors receive the complete P , it will construct the target executable E' by combining P with the old version executable E . This is demonstrated in Figure 10.

Developing different UCC compilation schemes. A modern compiler usually consists of a number of optimization and code generation passes such as partial redundancy elimination (PRE) [55], instruction selection, data allocation, and register allocation. In this dissertation, I pick data allocation and register allocation as two examples to study. When data allocation changes, all the load and store instructions that access the corresponding data need to change; when register allocation changes, all instructions that access the corresponding variables need to change. In particular, I integrate the update-conscious strategy as the last step of optimization. At this time, the compiler is ready to allocate each function's activation record and place all variable uses into registers. This assumption helps to analyze the properties of UCC in the dissertation. In practice, it may be combined with other optimization passes in different order, as I will discuss next.

Many other optimization passes may also benefit from UCC. For example, instruction selection determines the opcode of each binary instruction. When there are multiple choices, matching to the old version is preferred as it helps to improve the code similarity. I leave this as a future topic to be further explored and discussed in Chapter 7.

The best place for UCC in compilation. Figure 11 shows that other optimizations may still be performed after UCC. Part of the benefits exposed by UCC may be removed by later optimizations. For example, link-time optimization [56] may eliminate an unused branch, cause code shift once again, and thus reduce code similarity. The problem of finding the best place to integrate UCC for maximal code similarity falls in a more general problem in the compiler community, i.e., how to order different optimization passes to maximize the benefits. The problem has been intensively studied with both heuristic algorithms and exhaustive searching/pruning techniques proposed in the literature [40, 79, 81, 84]. In this dissertation, I assume UCC is the last optimization pass and leave the ordering problem as a future topic to explore.

UCC requires support from other analyses. Figure 11 also shows that UCC needs auxiliary information collected in early analysis passes and from other tools. For example,

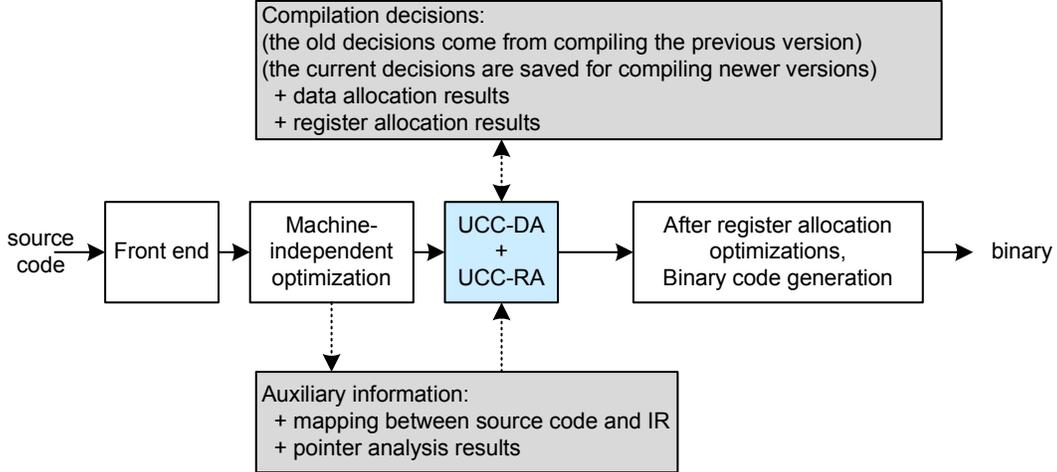


Figure 11: Placing update conscious compilation (UCC) in the traditional compilation flow.

TinyOS applications often use pointers and thus need pointer analysis to disambiguate memory accesses. The problem was studied by Cooper and Regehr in their `cXprop` tool [17]. They proposed to transform the `.c` code of TinyOS applications to CIL intermediate representation and track dataflow in the *pointer set* abstract domain, which serves as the basis for both must-alias and may-alias analyses. Their results showed good analysis precision [17] for constant propagation and other optimizations. The schemes proposed in this dissertation do not handle pointers directly and need to combine with a pointer analysis tool such as `cXprop` to ensure correctness.

Version control after multiple code updates. Currently UCC only focuses on improving the code similarity of consecutive code versions. After multiple code updates, there might exist multiple versions. Having multiple code versions is not a problem during compilation, since UCC is the last optimization pass in my framework. Other optimization passes do not consider old code, and perform exactly the same as before. When performing UCC, UCC only tries to minimize the difference from the preceding version.

At the network level, there may exist some sensors that do not have the required old version. For example, the current code update is from version 2.0 to 2.1 while some sensors only have version 1.0 (because they were either in the sleeping mode, or they were temporarily

disconnected from the network). This is a common problem for the diff-based patching strategy. The current solution is that sensors that do not have the required old code download the complete code image from their neighbors. This strategy will have insignificant impact on the overall energy efficiency: since code update is a high-priority task, every sensor needs to check its code version when it wakes up or re-connects to the network, and only a small number of sensors may skip an update completely.

3.2 UCC TECHNIQUES FOR GENERAL-PURPOSE APPLICATIONS

In this section, I will discuss the UCC schemes developed for general-purpose applications, i.e., the code to be executed on the main sensor processor. The schemes include a UCC register allocation scheme, a UCC data allocation scheme, and an integrated scheme that combines both. The goal is to generate the new binary image as similar to the old binary image as possible with minimal run-time performance loss.

3.2.1 UCC data allocation (UCC-DA) for general-purpose applications

The binary instructions may change due to changes in data allocation, e.g., relocating a variable requires updating the load/store instructions that access it. Increasing data allocation similarity tends to decrease such changes and improve the binary code similarity. However, there are tradeoffs that need to be carefully evaluated to ensure correctness.

3.2.1.1 Data allocation problem for general-purpose applications

The data allocation strategy can affect the binary level similarity, as illustrated in the example in Figure 12. In the original code (Figure 12(a)), three one-word 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 result as shown in Figure 12(b), in which all variables are assigned with new offsets, resulting in three different instructions. However,

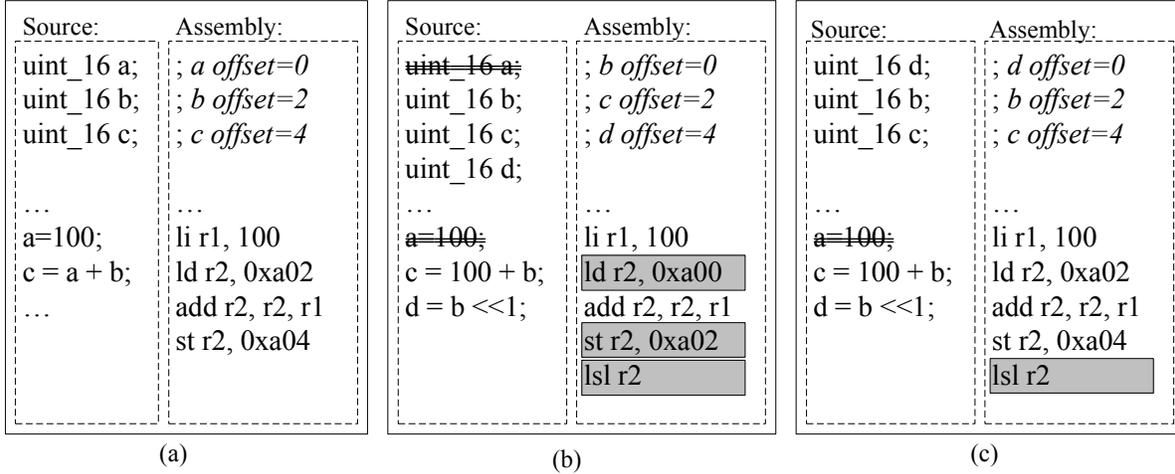


Figure 12: An update-conscious data allocation example. (a) Original source and assembly code; (b) New code and the changed instructions; (c) Incrementally generated new code with a smaller change.

an update-conscious algorithm should allocate the new variable `d` to `a`'s old location, as shown in Figure 12(c), resulting in only one different instruction.

However, keeping the data allocation the same as the old version may raise another problem. If there was no `d` in the new code and if the word taken by `a` was not claimed, we would require one more word in the function's activation record, and thus waste multiple RAM memory slots at runtime as the function can be invoked multiple times. This will increase the memory usage on remote sensors.

3.2.1.2 UCC data allocation for general-purpose applications

To address the problem of how to improve the data allocation similarity and keep the worst-case call stack size lower than the available RAM size, I propose a *threshold-based data allocation* mechanism [50]. The intuition is to reuse the space of the deleted variables, so that when there are more deleted variables than the new variables, it increases the memory usage only if the impact is acceptable.

- If there are more new variables than the deleted ones, the *threshold-based data allocation*

algorithm will first use up the space of the deleted variables and then allocate more space.

- If there are more deleted variables, some memory words in the corresponding function’s activation record will be left as “holes”. There are two options to save the space:
 - relocate some old variables to fill in the “holes”;
 - do not relocate.

The first option does not waste the runtime RAM space on the sensor node, but it needs to change the program code because of the variable relocation. The second option incurs fewer code changes but leaves “holes” in the call stack at runtime. A typical wireless sensor has only limited data memory, for example, 4KB RAM for Mica2 or MicaZ sensors. This data memory is used to store not only the call stack but also the data segment and the BSS segment. To ensure that the stack does not overflow, the total wasted RAM space should be less than a given threshold — `SpaceT`. Given a sensor application, I use the TinyOS tool `tos-ramsize` to collect the size of each segment and set the threshold `SpaceT` to be the free space left in RAM. A sample output of `tos-ramsize` is as follows.

```
$ tos-ramsize mica2 main.exe
BSS segment size is 1024, data segment size is 512
The upper bound on stack size is 2048
The upper bound on RAM usage is 3584
There are 512 unused bytes of RAM
```

It is straightforward that `SpaceT` should be set as

$$\text{SpaceT} = \text{SIZE}_{RAM} - \text{SIZE}_{Data_segment} - \text{SIZE}_{BSS} - \text{SIZE}_{Stack_footprint}$$

The detailed algorithm is shown in Algorithm 1. For clarity, the proposed algorithm elaborates on the procedures for variables of word type only. The principle can be similarly applied to other data types such as array and composite structures.

First, it collects the following profiles for each procedure $P_i (i \geq 0)$ in the program. In particular, $\text{Usage}_i(\mathbf{a})$ is from the code itself. NumOfInsts_i is from programmer’s expert knowledge assisted by program analysis tools, as I will explain later.

Algorithm 1 UCC-DA for general-purpose applications.

Input: Procedure list $P[]$, the wasted space threshold $SpaceT$.**Output:** The data allocation result.

```
1: for all  $P_i \in P[]$  do
2:   TotalWastedSpaceSize  $\leftarrow$  0;
3:   NumOfDelVi  $\leftarrow$  the total number of deleted variables in  $P_i$ ;
4:   NumOfNewVi  $\leftarrow$  the total number of new variables in  $P_i$ ;
5:   NumOfInstsi  $\leftarrow$  the projected maximal simultaneous instances of  $P_i$ ;
6:   if NumOfDelVi  $\leq$  NumOfNewVi then
7:     Reuses all the space from deleted variables;
8:     Allocate extra space to satisfy the remaining new variables;
9:   else
10:    Reuses all the space from deleted variables;
11:    ExtraSpaceSizei  $\leftarrow$  NumOfDelVi - NumOfNewVi;
12:    TotalWastedSpaceSize  $+=$  ExtraSpaceSizei  $\times$  NumOfInstsi;
13:   end if
14: end for
15: while TotalWastedSpaceSize  $>$  SpaceT do
16:   Max_Factor  $\leftarrow$  0;
17:   for  $P_i \in P[]$  AND ExtraSpaceSizei  $>$  0 do
18:     Usagei(last)  $\leftarrow$  the usage of the last variable in  $P_i$ ;
19:     Factori  $\leftarrow$   $\frac{NumOfInsts_i}{Usage_i(last)}$ ;
20:     if Factori  $>$  Max_Factor then
21:       Max_Factor  $\leftarrow$  Factori;
22:       To_Move  $\leftarrow$   $i$ ;
23:     end if
24:   end for
25:   Move the last variable in procedure To_Move to fill up a memory “hole”;
26:   TotalWastedSpaceSize  $- = 1 \times NumOfInsts_i$ ;
27: end while
```

NumOfDelV_i		the total number of deleted variables in P_i ;
NumOfNewV_i		the total number of new variables in P_i ;
NumOfInsts_i		the projected maximal simultaneous instances of P_i ;
$\text{Usage}_i(\mathbf{a})$		the usage of variable \mathbf{a} in P_i .

Second, it gradually allocates new variables within each procedure P_i as shown in Algorithm 1 line 6~13. Instead of removing the deleted variables directly, it only marks them as deleted variables so that their space can be reused by new variables. If NumOfNewV_i is larger than or equal to NumOfDelV_i , it reuses all the space from the deleted variables and allocates extra space to satisfy the remaining new variables. If NumOfNewV_i is smaller than NumOfDelV_i , i.e., new variables cannot reuse all space of the deleted ones, then it computes the number of words left to be filled using the following formula and moves to the next step.

$$\text{ExtraSpaceSize}_i = \text{NumOfDelV}_i - \text{NumOfNewV}_i \quad (3.1)$$

In this step, it adjusts the data allocation by incrementally relocating the variable that is allocated in the *last* memory slot in each procedure’s activation record. It keeps moving the last variable into a “hole” left by variable deletion, until the wasted memory space is smaller than the threshold. That is,

$$\sum_{\forall P_i} \text{ExtraSpaceSize}_i \times \text{NumOfInsts}_i \leq \text{SpaceT}. \quad (3.2)$$

The algorithm evaluates two factors at each step when determining which procedure should relocate its last variable to fill up one “hole”. One is the number of usages of its last variable $\text{Usage}_i(\text{last})$, and the other is the number of instances that the procedure can have on stack NumOfInsts_i .

Relocating a variable that is used more frequently causes more instruction updates, resulting in a larger patch and more transmission energy consumption. Thus, my algorithm tries to choose a rarely used variable to relocate in order to save transmission energy.

The other factor comes from the space constraint. A procedure that wastes one memory word in its activation record can waste five RAM slots if there are five active instances in the stack. As shown in Figure 13, for an event-driven environment such as TinyOS [76], interrupts use stack memory on top of the current running task. When non-atomic interrupt request (IRQ) is enabled, multiple instances of the same interrupt handler may stay in the

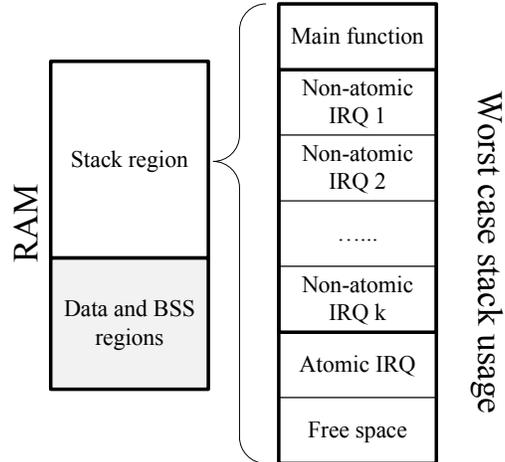


Figure 13: The sensor memory model.

stack. Therefore, my algorithm estimates the actual space wasted at runtime from each “hole”, and chooses the one that has the smallest number of instances.

Precisely tracking the stack usage is important to ensure safety, i.e., the worst-case stack size is still less than the RAM size. Two tools, `stack-estimator` [53] and `tos-ramsize` [67], have been developed in the literature to accomplish this goal. For example, when setting the verbosity level to 2, `tos-ramsize` prints out the worst-case depth of each interrupt vector, i.e, NumOfInsts_i (for each interrupt handler P_i).

```
$ tos-ramsize -verbosity 2 mica2 main.exe
... vector 12 max depth = 40 (not atomic)
...
```

The result indicates the interrupt handler 12 is non-atomic and its worst-case calling depth is 40. Wasting one word in interrupt handler 12’s activation record will waste 40 RAM memory slots at runtime. Currently `tos-ramsize` does not support the analysis of task functions. I take the approach in `stack-estimator` to build the call graph, and traverse the call graph to find the worst-case call count of each function, i.e., NumOfInsts_i (for each task function P_i). Similar to `tos-ramsize` and `stack-estimator`, recursion is not supported.

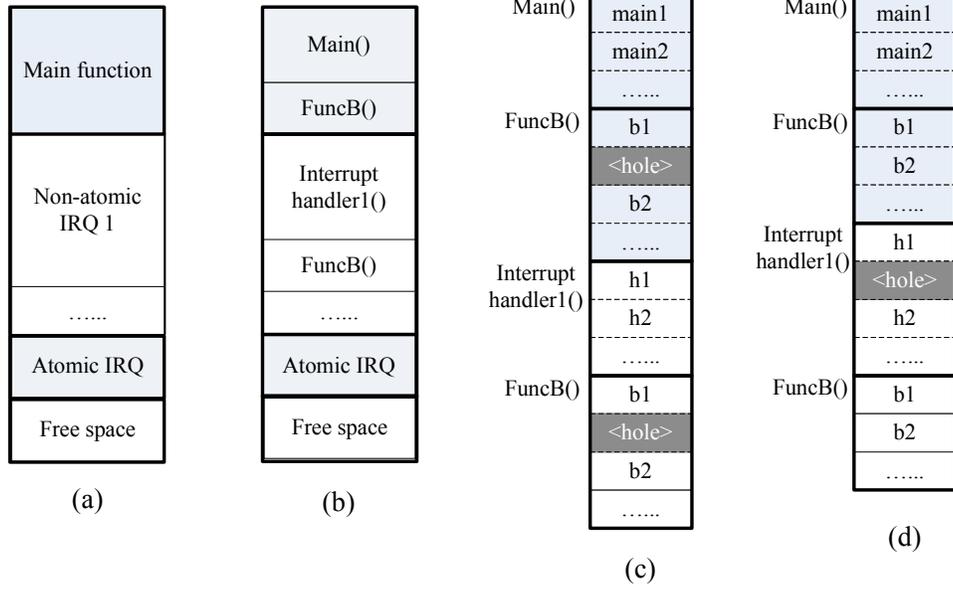


Figure 14: An example of data allocation for general-purpose applications.

When $Usage_i(\mathbf{a})$ (the usage of variable \mathbf{a} in P_i) and $NumOfInsts_i$ (the maximal instances of P_i) are ready, Figure 14 shows an example of how to identify the variable to relocate. $FuncB()$ is called by both $Main()$ and $Interrupt_handler1()$, so it has two instances on the stack, while the other procedures only have one instance each. Wasting one word in $FuncB()$ causes two memory slots to be wasted in RAM at runtime, while wasting one word in $Interrupt_handler1()$ causes only one wasted memory slot at runtime. Thus, in order to save RAM usage, my algorithm moves the last variable of $FuncB()$ instead of that of $Interrupt_handler1()$.

With such considerations, the selected procedure P_k should satisfy the following equation:

$$\frac{NumOfInsts_k}{Usage_k(\text{last})} = \text{MAX}\left(\frac{NumOfInsts_i}{Usage_i(\text{last})}\right) \quad (\forall i, \text{ExtraSpaceSize}_i > 0). \quad (3.3)$$

After deciding which procedure to pick, UCC-DA relocates the last variable in that procedure to one deleted memory word. By doing so, it can shrink the maximal runtime memory usage by $NumOfInsts_k$ (as it is the last variable in that procedure), and incur fewer code changes

(as the variable with fewer uses is selected). UCC-DA then decrements `ExtraSpaceSizek` and continues this step until Equation (3.2) is satisfied.

For the example in Figure 12, if `SpaceT=0` and `d` are not introduced, UCC-DA will reuse `a`'s space with `c`, which changes one instruction related to `c`. This still outperforms the default scheme in Figure 12(b), as `b` is not relocated.

3.2.2 UCC register allocation (UCC-RA) for general-purpose applications

The register allocation result, in addition to the data allocation result, can also affect the similarity between two binary images. In this section, I will discuss how to perform UCC register allocation when generating the new binary.

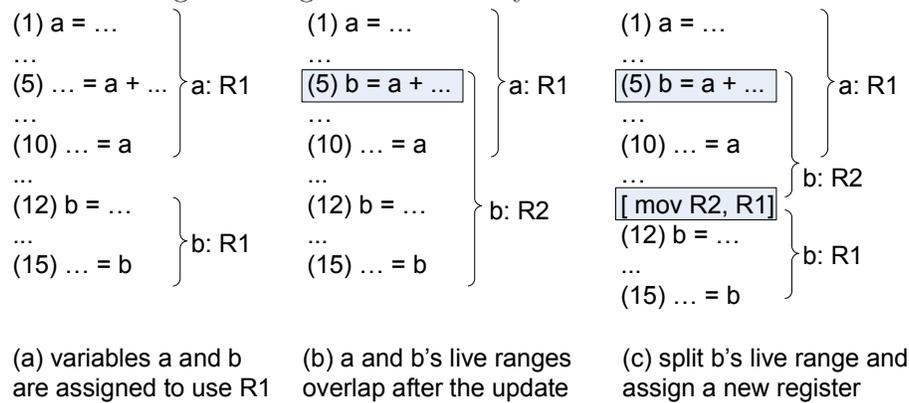


Figure 15: An example of register allocation for general-purpose applications.

3.2.2.1 Register allocation problem for general-purpose applications

Figure 15 illustrates why different register allocation decisions can greatly impact the code similarity, and therefore 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 15(a)). Assume a small code change extends `b`'s live range into `a`'s. If there are enough free registers, a traditional register allocator will assign different registers to them, as depicted in Figure 15(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 15(c). By comparing these two solutions, it is clear that solution (b) achieves faster code, and solution (c) results in less update cost. The discrepancy in energy consumption between data transmission and instruction execution makes solution (c) more appealing, as it consumes less energy unless the code is very frequently executed, or the update is extremely rare (as shown in Chapter 6).

3.2.2.2 UCC register allocation for general-purpose applications

The basic idea of UCC register allocation (UCC-RA) is that, when compiling a new version of code, it takes the register allocation result of the old version into consideration, and performs register allocation *with preference given to the allocation decisions for the old binary*.

To achieve this, IR instructions are first identified as “changed” or “non-changed”, and then successive instructions of the same type are grouped into chunks. The register allocator then allocates registers for each chunk. Decisions for “changed” chunks are made by UCC-RA, while decisions for “unchanged” chunks are taken from the old code before the update. For the variables whose live ranges span across the chunk boundary, the register allocation consistency is checked at the end. Inter-register movement instructions may be added to ensure semantic correctness.

While doing UCC-RA, each variable in the input chunk is tagged with the register name that was assigned to it in the old binary. This tag is called *preferred-register tag*. The *preferred-register tag* is a hint to improving code similarity in UCC-RA.

The register allocator then allocates registers for each changed chunk, and gradually matches the register assignment, or allocation decisions from both changed and non-changed 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 “non-changed” or vice versa, then the assignment in the “changed” chunk gives *preference* to the assignment in the “non-changed” 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 inter-register 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 inter-register movement instruction is inserted only if it is estimated to be energy-efficient.

Motivated by the 0/1 integer linear programming research for register allocation [29], the UCC-RA problem can be formulated as a non-linear integer programming problem. The general idea of how to select the decision variables and formulate the constraints and the objective function is addressed as follows.

The decision variables. I use a set of decision variables that represent the register assignments at each program point. A decision variable is defined as Equation 3.4.

The value of the decision variables $X_{op.a.s}^{Ri}$ can be 0 or 1. When register `Ri` is assigned to variable `a` at statement `s` for operation `op`, the value is 1; and otherwise, it is 0.

$$X_{op.a.s}^{Ri} = \begin{cases} 0 & : \text{Assertion is true.} \\ 1 & : \text{Assertion is false.} \end{cases} \quad (3.4)$$

<code>op</code>	Operation;
<code>a</code>	Variable;
<code>s</code>	Statement;
<code>Ri</code>	Register ($1 \leq i \leq 31$);

The operations can be classified into three categories: *def/use*, *load/store*, and *move* operations. Figure 16 shows a full list of decision variables for UCC-RA.

Def/Use decision variables. The *def/use* decision variables model the register assignment of a variable at one statement. When a variable is assigned a value at a statement, the statement is called a *def* point of the variable. I use decision variable $X_{def.a.s}^{Ri}$ to determine

Def/Use	$X_{def.a.s}^{Ri}$	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;
Load/Store	$X_{st.a.s}^{Ri}$	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 ;
Move	$X_{mov.out.a.s}^{Ri}$	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 ;

Figure 16: The decision variables used in UCC-RA.

whether to allocate variable \mathbf{a} to register \mathbf{R}_i at statement \mathbf{s} , if \mathbf{a} is defined at \mathbf{s} . If the value of this variable stays in this register afterward, i.e., it is not spilled to the memory or moved to another register, I set the value of decision variable $X_{cont.a.s}^{R_i}$ to 1.

When a variable is used at a statement, the statement is a *use* point of the variable. There can be two cases depending on whether this is the last use point of this variable or not. This information can be gathered through backward analysis. If it is the last use of this variable, there is no need to decide where to store the value after this statement, so we only need one decision variable to model the register assignment for the current statement. Decision variable $X_{lastUse.a.s}^{R_i}$ is used for this purpose. If the variable lives until later instruction, we use $X_{use.a.s}^{R_i}$ to model the register assignment for the current statement and $X_{useCont.a.s}^{R_i}$ to model the register assignment right after this statement. Based on the definition, one can see that $X_{use.a.s}^{R_i}$ and $X_{useCont.a.s}^{R_i}$ are exclusive from each other.

Load/Store decision variables. Usually we do not have enough registers to hold the values of all the live variables, so we need to store temporary unused values to cache or main memory and load them back to registers before their next uses. Because accessing memory is slower than accessing registers, memory spills increase the execution time.

I use the *load/store* decision variables to model memory spill decisions. If a variable \mathbf{a} is spilled from register \mathbf{R}_i to memory after statement \mathbf{s} , $X_{st.a.s}^{R_i}$ is set to 1. If a variable \mathbf{a} is loaded from memory to register \mathbf{R}_i before statement \mathbf{s} , $X_{ld.a.s}^{R_i}$ is set to 1. These two decision variables model not only whether to load/store the value of a variable from/to memory but also the register assignment after/before the load/store.

Decision variable $X_{cont.a.s}^{mem}$ is then used to model whether the value of a variable exists in memory or not. It is set to be 0 until the value of variable \mathbf{a} is stored to memory.

Move decision variables. As shown in Figure 15, when performing update-conscious compilation, `mov` instructions may be added to keep the register allocation similar to the old version. These `mov` instructions are considered as run-time overhead also.

To model this overhead, I use the *move* decision variables. Intuitively, I can also use one decision variable $X_{mov.a.s}^{R_i \leftarrow R_j}$ to represent whether to move the value of \mathbf{a} from \mathbf{R}_j to \mathbf{R}_i at statement \mathbf{s} . The reason that I use two decision variables $X_{mov.in.a.s}^{R_i}$ and $X_{mov.out.a.s}^{R_i}$ instead, is that the former design will introduce too many decision variables. Let us 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, I simply combine the corresponding move-in and move-out variables to implement the register move.

An example. For the code chunk in Figure 15(a), I first introduce a set of decision variables that represent the register assignments that we need to make at each program point. For example, if variable `a` is allocated in register `R1` at statement (1), then we have $X_{def.a.1}^{R1} = 1$ and $\forall Ri, Ri \neq R1, X_{def.a.1}^{Ri} = 0$. As another example, if we decided to insert an instruction “`mov R2 to R3`” for `b` before statement (4), I set $X_{mov.out.b.4}^{R2} = 1$, $X_{mov.in.b.4}^{R3} = 1$, and all other *move* 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 constraints. With the defined decision variables, I convert the register allocation problem into a problem of finding the 0/1 solution 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.

Two symbols. In the following discussion, two symbols $U_{a.s}^{Ri}$ and $V_{a.last_s}^{Ri}$ are introduced to simplify the equations. $U_{a.s}^{Ri}$ is defined to describe the register assignment of variable `a` at instruction `s`. Based on whether the statement is a *def* or *use* point of the variable, the value of $U_{a.s}^{Ri}$ can be formulated using the following equations:

$$U_{a.s}^{Ri} = \begin{cases} X_{def.a.s}^{Ri} & : \text{ if } a \text{ is defined at } s \\ X_{use.a.s}^{Ri} + X_{useCont.a.s}^{Ri} & : \text{ if } a \text{ is used at } s \\ X_{lastUse.a.s}^{Ri} & : \text{ if } a \text{ is used and dies at } s \end{cases} \quad (3.5)$$

$V_{a.last_s}^{Ri}$ is defined to show the register assignment of variable `a` after the most recent access point (`last_s`). The reason we want to introduce this symbol is that the register assignment at a former instruction affects the register assignment at the later instruction. It is preferable to store the whole life range of a variable in one register, in order to avoid

the extra *load*, *store*, or *move* instructions. Depending on whether instruction `last_s` is a *def* point or *use* point of variable `a`, $V_{a.last_s}^{Ri}$ is calculated in different ways.

$$V_{a.last_s}^{Ri} = \begin{cases} X_{cont.a.last_s}^{Ri} & : \text{ if } a \text{ is defined at } last_s \\ X_{useCont.a.last_s}^{Ri} & : \text{ if } a \text{ is used at } last_s \end{cases} \quad (3.6)$$

The defined constraints are presented as shown below.

One register occupation. Each variable should be allocated to one and only one register at its *def* or *use* point (Equation 3.7). This includes three cases: (i) if a variable `a` is at its *def* point `s`, one and only one $X_{def.a.s}^{Ri}$ can be 1 (Equation 3.8); (ii) if a variable `a` is at a *use* point and this is the last *use* for the variable, one and only one $X_{lastUse.a.s}^{Ri}$ can be 1 (Equation 3.9); (iii) if a variable `a` is at a *use* but not the last *use* point, the actual register assignment can be represented by $X_{use.a.s}^{Ri} + X_{useCont.a.s}^{Ri}$ (because they are exclusive) and should satisfy the constraint in Equation 3.10.

$$\sum_{\forall Ri} U_{a.s}^{Ri} = 1 \quad (3.7)$$

$$\sum_{\forall Ri} X_{def.a.s}^{Ri} = 1 \quad (3.8)$$

$$\sum_{\forall Ri} X_{lastUse.a.s}^{Ri} = 1 \quad (3.9)$$

$$\sum_{\forall Ri} (X_{use.a.s}^{Ri} + X_{useCont.a.s}^{Ri}) = 1 \quad (3.10)$$

Inter-register moves. To ensure valid inter-register movements, I define constraints on *move* decision variables. Equation 3.11 indicates that for a variable `a` at each program point `s`, a move instruction may and may not be inserted, and the move-in and move-out decision variables must appear in pairs.

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

$$\sum_{\forall Ri} X_{mov.out.a.s}^{Ri} = \sum_{\forall Ri} X_{mov.in.a.s}^{Ri} \quad (3.11)$$

Memory spill related. At a statement `s`, variable `a` may be loaded from the memory or may come from inter-register movement. Equation 3.12 indicates that 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.

$$\begin{aligned}
X_{st.a.s}^{Ri} &\leq X_{def.a.s}^{Ri} + X_{mov.in.a.s}^{Ri} \\
X_{mov.out.a.s}^{Ri} &\leq X_{def.a.s}^{Ri} \\
X_{cont.a.s}^{Ri} &\leq X_{def.a.s}^{Ri} + X_{mov.in.a.s}^{Ri}
\end{aligned} \tag{3.12}$$

Equation 3.13 indicates that for the code spill at a *def* point, only a store instruction may be possibly generated.

$$X_{cont.a.s}^{mem} \leq \sum_{\forall Ri} X_{st.a.s}^{Ri} \tag{3.13}$$

At a *use* point, a variable may be located in a register due to its use in the previous instruction, or loaded from the memory, or moved from another register. Depending on whether the *use* is the last use point, Equation 3.14 defines the following two constraints:

$$\begin{aligned}
X_{use.a.s}^{Ri} + X_{useCont.a.s}^{Ri} &\leq X_{cont.a.(s-1)}^{Ri} + X_{ld.a.s}^{Ri} + X_{mov.in.a.s}^{Ri} \\
X_{last.a.s}^{Ri} &\leq X_{cont.a.(s-1)}^{Ri} + X_{ld.a.s}^{Ri} + X_{mov.in.a.s}^{Ri}
\end{aligned} \tag{3.14}$$

Equation 3.15 defines the constraint that either a load spill or an inter-register movement before the *use* point is needed.

$$\begin{aligned}
\sum_{\forall Ri} X_{ld.a.s}^{Ri} &\leq X_{cont.a.(s-1)}^{mem} \\
\sum_{\forall Ri} X_{mov.out.a.s}^{Ri} &\leq X_{cont.a.(s-1)}^{Ri}
\end{aligned} \tag{3.15}$$

No register assignment conflict. The following constraints are used to ensure that one register is assigned to only one variable at a time.

Equation 3.16 defines the constraint that there is no register assignment conflict between the current variable and the active variables that are processed before.

$$\text{For each } Ri, \sum_{\forall var} V_{var.last.s}^{Ri} + U_{a.s}^{Ri} \leq 1 \tag{3.16}$$

For example, the following constraints are generated at statement (2) in Figure 15:

$$\begin{aligned}
U_{b.2}^{Ri} &= X_{def.b.2}^{Ri} \\
V_{a.last_use}^{Ri} &= X_{cont.a.1}^{Ri} \\
\text{Equation 3.16} \Rightarrow X_{def.b.2}^{Ri} + X_{cont.a.1}^{Ri} &\leq 1
\end{aligned} \tag{3.17}$$

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

$$\text{For each } Ri, \sum_{\forall var} U_{var.s}^{Ri} \leq 1 \tag{3.18}$$

For example, the following constraints are generated at statement (6) in Figure 15:

$$\begin{aligned}
U_{a.6}^{Ri} &= X_{lastUse.a.6}^{Ri} \\
U_{b.6}^{Ri} &= X_{use.b.6}^{Ri} + X_{useCont.b.6}^{Ri} \\
\text{Equation 3.18} \Rightarrow X_{lastUse.a.6}^{Ri} + X_{use.b.6}^{Ri} + X_{useCont.b.6}^{Ri} &\leq 1
\end{aligned} \tag{3.19}$$

Continuous register assignment. For Mica2 micro controllers, UCC-RA needs to enforce another type of constraint. Each register in Mica2 has 8 bits, i.e., one byte. A 32-bit integer variable takes 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:

$$\begin{aligned}
X_{use.(a).s}^{Ri} &= X_{use.(a+1).s}^{Ri+1} \\
X_{use.(a+1).s}^{Ri} &= X_{use.(a+2).s}^{Ri+1} \\
X_{use.(a+2).s}^{Ri} &= X_{use.(a+3).s}^{Ri+1}
\end{aligned} \tag{3.20}$$

At the boundary of changed and unchanged code chunks, and at the merge point of control flows, UCC-RA inserts inter-register move instructions to make sure that the values are in proper registers before their next uses. In my future work, instead of performing inter-register movements, I will introduce constraints similar to those in [29] for the merge point of control flows.

The objective function. The goal of my integer programming is to minimize the objective function on total energy consumption, as expressed in Equation 3.21 in Figure 17.

The equation defines the total energy consumption of the changed IR chunk under different register allocation decisions. The notations used in equation (3.21) are listed in Figure 18. Other terms are explained as follows.

$$\mathbf{E}_{total} = \text{chg}(\mathbf{s}) \times \mathbf{E}_{changed_IR} + (1 - \text{chg}(\mathbf{s})) \times \mathbf{E}_{unchanged_IR} + \mathbf{E}_{spill} + \mathbf{E}_{extra} \quad (3.21)$$

where

$$\mathbf{E}_{changed_IR} = \sum_{\forall s} (\text{freq}(\mathbf{s}) \times \mathbf{E}_{exe}) + \sum_{\forall s} (\mathbf{E}_{trans}) \quad (3.22)$$

$$\mathbf{E}_{unchanged_IR} = \sum_{\forall s} (\text{freq}(\mathbf{s}) \times \mathbf{E}_{exe}) + \sum_{\forall s} (1 - \prod_{\forall a} \mathbf{X}_{def/use.a.s}^{prefer(a,s)}) \times \mathbf{E}_{trans} \quad (3.23)$$

$$\begin{aligned} \mathbf{E}_{spill} = & \sum_{\forall s,a,Ri} (\text{freq}(\mathbf{s}) \times (\mathbf{X}_{st.a.s}^{Ri} + \mathbf{X}_{ld.a.s}^{Ri}) \times \mathbf{E}_{exe}) + \\ & \sum_{\forall s,a,Ri} ((1 - \text{spill}(a, Ri, \mathbf{s})) \times (\mathbf{X}_{ld.a.s}^{Ri} + \mathbf{X}_{st.a.s}^{Ri}) \times \mathbf{E}_{trans}) \end{aligned} \quad (3.24)$$

$$\begin{aligned} \mathbf{E}_{extra} = & \sum_{\forall s,a,Ri} (\text{freq}(\mathbf{s}) \times \mathbf{X}_{mov.in.a.s}^{Ri} \times \mathbf{E}_{exe}) + \\ & \sum_{\forall a,s,Ri} (\mathbf{X}_{mov.in.a.s}^{Ri} \times \mathbf{E}_{trans}) \end{aligned} \quad (3.25)$$

Figure 17: The objective function used in UCC-RA.

\mathbf{E}_{spill} specifies the energy consumption due to code spill. It includes two components: the execution energy and the dissemination energy. The former depends on the code quality which is the main goal of many traditional 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, i.e., either $(1 - \text{spill}(a, Ri, \mathbf{s})) = 0$ or $(\mathbf{X}_{ld.a.s}^{Ri} + \mathbf{X}_{st.a.s}^{Ri}) = 0$ in the Equation 3.24. For example, if a is spilled to $R1$ in both new and old binaries, then we have zero transmission cost.

$$\text{for } R1, 1 - \text{spill}(a, R1, \mathbf{s}) = 0, \mathbf{X}_{ld.a.s}^{R1} + \mathbf{X}_{st.a.s}^{R1} = 1$$

$$\text{for } Ri (Ri \neq R1), 1 - \text{spill}(a, Ri, \mathbf{s}) = 1, \mathbf{X}_{ld.a.s}^{Ri} + \mathbf{X}_{st.a.s}^{Ri} = 0$$

$\mathbf{E}_{changed_IR}$ 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

E_{trans}	the energy consumed to disseminate one instruction in WSN;
E_{exe}	the energy consumed to execute one instruction. We use the averaged 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 counter 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;

Figure 18: The notation used in the UCC-RA objective function.

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 there is 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, i.e., the same binary instruction “ $add\ R1, R2$ ” is generated. If a is assigned to a different register, say $R3$ such that “ $add\ R3, R2$ ” is generated, then this new instruction needs to be disseminated to replace the old one on the sensor. As shown in Equation 3.23, this component is non-linear – 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 inter-register movements. This term is zero if a traditional compiler decision is used. My UCC-RA targets achieving overall energy efficiency, i.e., E_{extra} is positive only when it can gain more reduction from other components, e.g., $E_{unchanged_IR}$.

In the above model, X_*^* are decision variables that need to be determined by the UCC-

RA, while others (such as `chg(s)`, `freq(s)`) are known for a given code chunk. Since Equation 3.23 is non-linear, the above formulation of UCC-RA results in a mixed integer non-linear programming problem (MINLP) [10]. While the speed of MINLP solvers has been improved greatly in recent years [10], it is still much slower than solving a linear problem. My experiments showed that MINLP can be orders of magnitude slower than a linear problem of similar size, i.e., similar number of decision variables and constraints. Next I will discuss how to convert the MINLP problem to an ILP problem through approximation.

Solve an ILP problem. In this section I 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 my model has at most two different operands), assume their preferred-registers are R1 and R2 respectively. The energy consumption can be modeled as

$$\sum_{\forall s} (1 - X_{use.a...}^{R1} + 1 - X_{use.b...}^{R2}) \times E_{trans} \times \delta \quad (3.26)$$

where $\delta = 3/4$ is a coefficient that approximates the update cost. δ 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:

- `a` is in R1, `b` is in R2;
- `a` is in R1, `b` is not in R2;
- `a` is not in R1, `b` is in R2;
- `a` is not in R1, `b` is not in R2.

It is clear that the first case has no update cost, while each of other three cases needs to update one instruction. Therefore, the averaged update cost is $(3/4) \times Cost_{single}$, which makes δ to be $3/4$.

After converting the model into an ILP problem, I chose a widely used ILP solver — `LP_solve` [8] to find the optimal assignment to the decision variables such that the energy

cost in the objective function is minimized. The decision variables are then mapped back to register assignments such that the new binary and the update script can be generated.

The UCC-RA heuristic. UCC-RA is a greedy heuristic algorithm. By applying ILP formulation only to the changed chunks, UCC-RA strives to achieve overall energy efficiency for updating the whole program. In my framework, UCC-RA can always reach the global optimum because: (i) the compilation decisions for the changed chunks are optimal or near-optimal due to ILP formulation; (ii) the compilation decisions for the unchanged chunks are optimal also — since there is no need to update the unchanged code, the transmission energy is minimized; since UCC-RA is assumed to be the last optimization pass, the execution energy consumption of the unchanged code is also optimal. Given that UCC achieves the optimal solutions for both changed and unchanged chunks, the overall energy efficiency is optimal.

When UCC-RA is not the last optimization pass, the result may not be globally optimal. For example, further optimizing the UCC-identified unchanged code may reduce its execution energy consumption. If the code is frequently executed, a globally optimal strategy should decide to optimize the code, and pay the one-time dissemination energy to save long-time execution energy consumption. As we discussed in Section 3.1, finding the best place to integrate update-conscious compilation falls in a more general problem, i.e., the optimization ordering problem [40, 81]. I leave it as an open problem for future research.

3.2.3 The integration of UCC-DA and UCC-RA

The preceding section considers register allocation only. When evaluating energy consumption, the changed and unchanged IR statements are treated differently. For the changed IR statements, the code update cannot be avoided. Thus, using the UCC-RA scheme does not help. For the unchanged IR statements, keeping the same register assignment improves the code similarity and reduces the update script size.

However, UCC-DA and UCC-RA interfere with each other — if some variables are relocated, their corresponding load and store instructions have to be updated, no matter if the register allocation chooses the same allocation as the previous version or not. Therefore, in the object function, the energy consumption components of these load and store instructions

should be modeled as changed instructions.

Based on this observation, I propose two approaches to integrate UCC-DA and UCC-RA schemes. The first approach is to represent UCC-DA as a set of new constraints in the previously formulated ILP problem, and solve both UCC-DA and UCC-RA in one step. The second approach is to solve UCC-DA and UCC-RA in two sequential steps.

3.2.3.1 Performing UCC-DA and UCC-RA in one step

In order to integrate UCC-DA and UCC-RA and solve in one step, I will first introduce a new decision variable to model the data allocation decision, and then define the related constraints and the revised objective function.

Decision variables. A new decision variable x_a^{move} is defined to determine whether to relocate variable a in memory. The variable is set to 0 if the variable a is kept in the same location as in the old code, and is set to 1 otherwise.

Data allocation related constraints. Let $ExtraSpaceSize_i$ represent the wasted memory space for one instance of procedure P_i . As described in the UCC-DA algorithm 1, the total wasted space on the call stack cannot go beyond the threshold $SpaceT$. This constraint is formulated as the following Equation 3.27.

$$\sum_{\forall P_i} ExtraSpaceSize_i \times InstsNum_i \leq SpaceT \quad (3.27)$$

The wasted memory space for one instance of procedure P_i is presented in the following equation, which is equal to the number of deleted variables minus the number of new variables, then minus the number of relocated variables.

$$ExtraSpaceSize_i = \begin{cases} 0 & : if(NumOfDelV_i \leq NumOfNewV_i) \\ NumOfDelV_i - NumOfNewV_i - \sum_{\forall a} x_a^{move} & : otherwise \end{cases} \quad (3.28)$$

The variables defined in a procedure should be relocated in back order, i.e., the last variable of this procedure is always relocated before the-last-but-one. This constraint is formulated as Equation 3.29.

$$\forall a, b \quad old_addr(a) \leq old_addr(b) \Rightarrow x_a^{move} \leq x_b^{move} \quad (3.29)$$

Integrated objective function. The objective function of UCC-RA (Equation 3.21) contains four parts. The energy consumption of the *ALU* instructions is formulated as $E_{changed_IR}$ and $E_{unchanged_IR}$. The energy consumption of the *spill* instructions is formulated as E_{spill} . The energy consumption of the inserted *mov* instructions is formulated as E_{extra} . Because the data allocation result will only affect the transmission energy consumption of the load/store instructions, only E_{spill} needs to be adjusted.

In UCC-RA, whether a load/store instruction needs to be sent to remote sensors (i.e., needs transmission energy) depends on two factors: whether there was a spill in the old binary ($\text{spill}(\mathbf{a}, \mathbf{Ri}, \mathbf{s})$), and how the variables are allocated in the registers ($X_{ld.a.s}^{Ri}$ and $X_{st.a.s}^{Ri}$). For the instructions that did not have the register spill in the old binary, the code update is required, because this load/store instruction is a new instruction. Otherwise, it depends on whether the register allocation results are the same as the old ones. In short, only when $X_{ld.a.s}^{Ri}$ (or $X_{st.a.s}^{Ri}$) and $\text{spill}(\mathbf{a}, \mathbf{Ri}, \mathbf{s})$ are set to be “1”, can the transmission energy be saved.

$$\begin{aligned}
E_{spill} = & \sum_{\forall s, a, Ri} \text{freq}(\mathbf{s}) \times (X_{st.a.s}^{Ri} + X_{ld.a.s}^{Ri}) \times E_{exe} + \\
& \sum_{\forall s, a, Ri} (1 - \text{spill}(\mathbf{a}, \mathbf{Ri}, \mathbf{s})) \times (X_{ld.a.s}^{Ri} + X_{st.a.s}^{Ri}) \times (1 - X_a^{move}) \times E_{trans} \quad (3.30)
\end{aligned}$$

Figure 19: The objective function used in ILP-based UCC integration.

When UCC-DA is also considered, whether a load/store instruction needs to be sent also depends on the location of the variable. If the variable gets relocated, i.e., $X_a^{move}=1$, then the instruction has to be updated. In other words, the instruction update is not necessary only when the following three conditions are all satisfied:

- $\text{spill}(\mathbf{a}, \mathbf{Ri}, \mathbf{s})$ is set to be “1”, which means this instruction was a *spill* instruction in the old binary;
- $X_{ld.a.s}^{Ri}$ or $X_{st.a.s}^{Ri}$ is set to “1”, which means the new register allocation result is the same as the old binary; and

- X_a^{move} is set to “0”, which means the variable is not relocated in memory.

Thus, the energy consumption of the load/store instructions is re-modeled in Equation 3.30.

Due to the introduction of X_a^{move} , the second term of E_{spill} in Equation(3.30) is not linear. That is, the problem becomes a MINLP [10] problem again. To save the time spent in solving the problem, I convert it into an ILP with approximation — similar to section 3.2.2.2, E_{spill} in Equation 3.30 can be rewritten as Equation 3.31 (δ is set as 3/4).

$$E_{spill} = \sum_{\forall s,a,Ri} \text{freq}(\mathbf{s}) \times (X_{st.a.s}^{Ri} + X_{ld.a.s}^{Ri}) \times E_{exe} + \sum_{\forall s,a,Ri} (1 - \text{spill}(\mathbf{a}, Ri, \mathbf{s}) \times (X_{ld.a.s}^{Ri} + X_{st.a.s}^{Ri} + 1 - X_a^{move}) \times \delta) \times E_{trans} \quad (3.31)$$

Figure 20: The converted objective function used in the ILP based UCC integration.

3.2.3.2 Performing UCC-DA and UCC-RA in two steps

Integrating UCC-DA constraints as above requires introducing a decision variable (X_a^{move}) for each variable, and N in total, where N is the number of variables used in the program. This increases the complexity of the ILP problem, and also the time to solve it. In this section, I design a two-step heuristic that performs the data allocation and the register allocation in two sequential steps. It does UCC-DA first and identifies all relocated variables. This information is passed to UCC-RA, in which all memory access statements that access relocated variables are considered as changed. This scheme does not affect the complexity of the ILP problem for UCC-RA.

`datachg(s)` | if statement \mathbf{s} is a memory access that accesses a relocated variable.

Figure 21: The notation used in the two-step integration.

The detailed algorithm is described below. I introduce `datachg(s)` (Figure 21) to describe whether an unchanged IR statement needs update because of data relocation. After

finishing UCC-DA, the compiler marks all relocated variables and sets their values to the `datachg(s)` for each memory access statement `s`.

$$\begin{aligned}
 E_{spill} = & \sum_{\forall s,a,Ri} \text{freq}(s) \times (X_{st.a.s}^{Ri} + X_{ld.a.s}^{Ri}) \times E_{exe} + \\
 & \sum_{\forall s,a,Ri} (1 - \text{spill}(a, Ri, s) \times (X_{ld.a.s}^{Ri} + X_{st.a.s}^{Ri}) \times (1 - \text{datachg}(s))) \times E_{trans} \quad (3.32)
 \end{aligned}$$

Figure 22: The objective function used in the two-step approach.

The objective function needs to be updated as shown in Equation 3.32. The only difference between Equation 3.32 and Equation 3.30 (the objective function of the one-step approach) is that `datachg(s)` is a constant when doing UCC-RA while x_a^{move} is a decision variable.

The two-step approach is easy to implement, however the result may be sub-optimal. For example, UCC-DA needs to relocate one of two variables `a` and `b`. Since variable `a` is more frequently used than `b`, UCC-DA decides to relocate variable `b`, which is predicted to generate fewer code updates. But if later UCC-RA decides to assign another register for `a`, all variable `a`-related instructions must be updated. A global energy-efficient decision should relocate variable `a` instead of `b`, which can be achieved in the one-step integrated ILP formulation. While the ILP formulation produces a better solution, it takes longer to solve. I will evaluate the tradeoff in the experiment section.

3.3 UCC TECHNIQUES FOR DSP APPLICATIONS

As discussed in Chapter 2, when a sensor application decides to offload part of its code to the DSP co-processor, it should take advantage of the specific architectural features to speed-up the processing. The auto-addressing mode supported by the address generation unit (AGU) enables the parallel computation of address registers, which helps reduce the code size and improve performance. Different offset assignment heuristics can be applied to

achieve optimal data allocation and address register allocation. In this section, I will build my algorithms on top of existing heuristics [51, 59, 86].

Due to the close connection of data allocation and code generation for DSP applications, improving the data allocation similarity helps keep the addressing mode selections of many instructions similar to those in the old version, and improves the binary similarity. Therefore, an update-conscious data allocation scheme is desired for DSP application updates.

Usually, a DSP chip has multiple address registers, which creates an address register allocation problem. Current works [51, 59, 86] let different address registers handle different subsets of variables. For small to medium level code updates, while it is possible to re-partition the variables that should be handled by each address register, my experimental results showed that keeping the old variable partition for address registers is beneficial. For large changes, it is better to re-allocate from scratch. More details will be elaborated in Chapter 6.

3.3.1 Data allocation problem for DSP applications

A motivational example of the proposed UCC data allocation (UCC-DA) scheme is shown in Figure 23. When a DSP application has a small update, the binary code after the change will preferably be similar to the one before the update. However, update-oblivious schemes generate the new memory layout and its corresponding binary code without considering the similarity between different versions.

Figure 24 illustrates the data allocation result of the example shown in Figure 23 using an update-oblivious algorithm (CSOA [59]). Figure 24(a) shows the new code after a simple change in the above example, i.e., the third instruction is changed (in the box). Using the new access graph and interference graphs, CSOA generates a very different variable coalescing result (Figure 24(d)). The memory layout difference further translates to selecting different addressing instructions at each memory access (Figure 25).

However, an update-conscious algorithm would read in the old access graph and its interference graph, and generate a new memory layout that can help minimize the difference between the new and old binaries and thus the update script. For the given example,

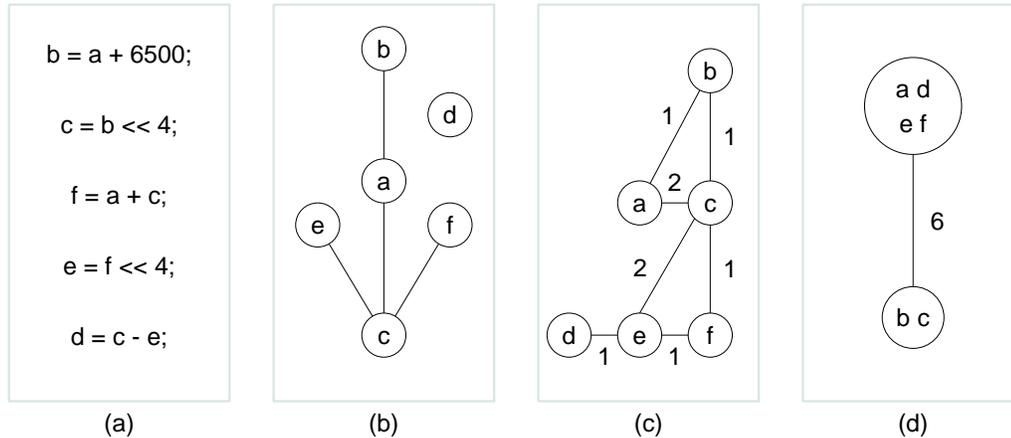


Figure 23: An example of data allocation for DSP applications. (a) The IR code; (b) The access graph; (c) The interference graph; (d) The data allocation result.

it would generate the same memory layout as the old version (Figure 23(d)). Figure 25 compares the update-oblivious and update-conscious data allocation algorithms. Using the update-oblivious scheme, four of seven instructions need to be updated due to the data allocation change, and using the update-conscious scheme completely avoids the update to these four instructions.

3.3.2 UCC data allocation (UCC-DA) for DSP applications

The design goal of UCC data allocation is to improve data allocation similarity between the old and new versions, and minimize run-time performance loss. To reach this goal for DSP applications, I design an incremental coalescing offset assignment scheme that includes ICSSOA for simple offset assignment (with one address register) and ICGOA for general offset assignment (with multiple address registers).

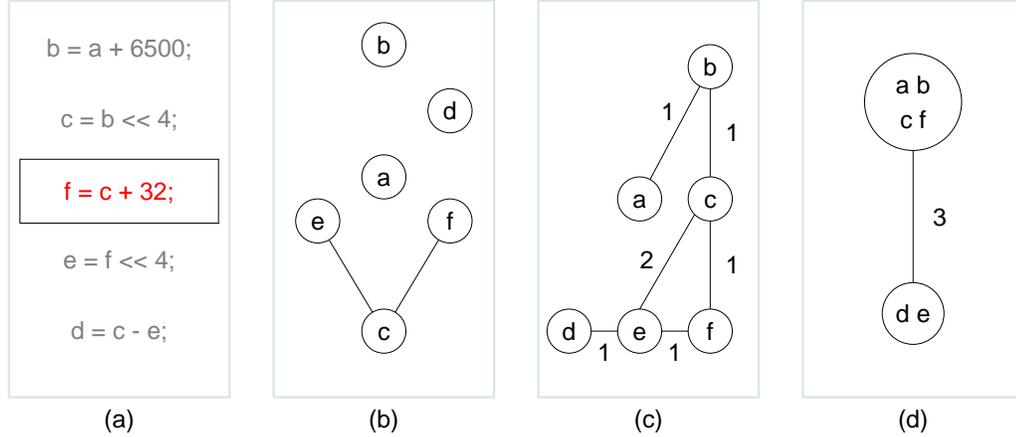


Figure 24: A motivational example of the need for UCC data allocation for DSP applications. (a) The C code after a simple update over Figure 23; (b) The new interference graph; (c) The new access graph; (d) The new data allocation using CSOA, showing significantly different layout from the original assignment shown in Figure 23(d).

3.3.2.1 Coalescing single offset assignment (CSOA)

Since my ICSOA scheme is based on an existing update-oblivious scheme (CSOA) [87], I will discuss CSOA in this section. Given a piece of code compile, CSOA takes its variable access graph and interference graph as input. In an *access graph*, a vertex represents a variable, an edge represents a sequential access between the two related variables, and the weight of an edge represents the count of such sequential accesses. In an *interference graph*, a vertex represents a variable, and an edge indicates if the live ranges of the two related variables overlap.

CSOA [87] employs an iteration-based heuristic to find the *maximum weight path cover* (MWPC) of the *access graph* while coalescing the variables that do not interfere with each other. In each iteration, CSOA selects one vertex, and based on pre-defined cost-saving functions, either allocates a new memory slot to it, or coalesces it with an allocated slot. When CSOA terminates, the MWPC of all coalescing groups determines the data allocation result. The variables coalesced in one group share the same memory slot.

	Access sequence	Original code	Update-Oblivious		Update-Conscious	
			code	update	code	update
0	a	•++	•	diff**	•++	
1	b	•	•		•	
2	b	•	•		•	
3	c	•- -	•	diff	•	diff
4	→ a*	•++		diff		diff
5	c	•- -	•	diff	•- -	
6	f	•	•		•	
7	f	•	•++	diff**	•	
8	e	•++	•- -	diff**	•++	
9	c	•- -	•++	diff**	•- -	
10	e	•	•		•	
11	d	•	•		•	

*: This access only exists in the old version.

** : The instruction that needs to be updated, due to data allocation changes.

•++ : An instruction with post-increment addressing.

•- - : An instruction with post-decrement addressing.

The old version memory layout is “slot 0: a, d, e, f; slot 1: b, c”

The memory layout generated by the update-oblivious scheme is:

“slot 0: a, b, c, f; slot 1: d, e”.

The memory layout generated by the update-conscious scheme is:

“slot 0: a, d, e, f; slot 1: b, c”.

Figure 25: The update script comparison between CSOA and the update-conscious scheme.

3.3.2.2 Incremental coalescing single offset assignment (ICSOA)

To minimize the update script, I propose to perform update-conscious code update through incremental coalescing SOA (ICSOA) (Figure 26). When a DSP application undergoes a small update on the server side, ICSOA reads in the old access graph and its interference graph, and strives to generate a new memory layout that minimizes the update script. On the mobile system side, only the update script needs to be downloaded. With simple interpretation, the mobile system regenerates the new binary and/or the new memory layout.

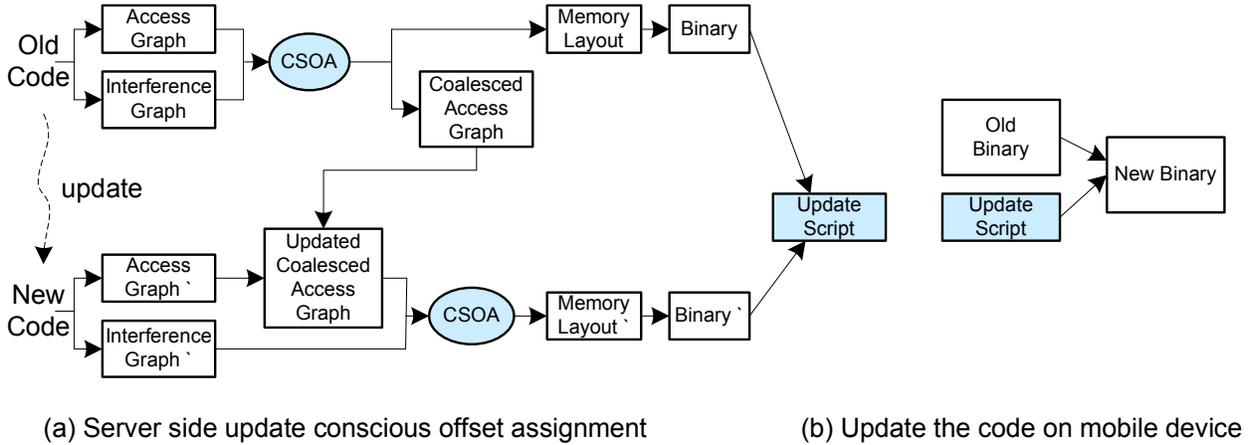


Figure 26: An overview of the ICSOA-based code update scheme.

The pseudo code of ICSOA is shown in Algorithm 2. The basic idea is to incorporate the code changes to the old access graph, and iteratively find the MWPC while coalescing variables using the updated interferences.

Function `update_access_graph()`. It combines the coalescing access graph of the old version (CAG_1) and the access graph after update (AG_2) into a new access graph (AG_{NEW}). ICSOA builds AG_{NEW} based on CAG_1 , by adding new variable nodes and removing unused nodes, so that AG_{NEW} not only represents the updated access sequence but also keeps all the coalescing offset assignment result from the old version. Using AG_{NEW} instead of AG_2 as the offset assignment input helps to improve the offset assignment similarity with the previous version, and reduces the patch transmission overhead. However, when the code change is relatively large, the energy saved by improving code similarity may be offset by

the performance loss (due to sub-optimal code generated from ICSOA). For this reason, when combining the graphs, `update_access_graph()` evaluates the number of accesses of each old variable in the new code, and extracts it from its coalescing group if the variable has more new or updated accesses than the unchanged ones. The intuition is to extract the variables from their old coalescing groups only if it can bring explicit benefits. A new node is introduced for each extracted variable; empty coalescing groups are removed from AG_{NEW} . At the end, the function adjusts the weights of impacted access edges accordingly to finish the update.

Algorithm 2 Incremental coalescing-based SOA (ICSOA)

Input: AS_1, AS_2 : access sequences before and after update;

IG_1, IG_2 : interference graphs before and after update;

Output: the offset assignment.

- 1: $AG_1 \leftarrow$ Build access graph using AS_1 ;
 - 2: $AG_2 \leftarrow$ Build access graph using AS_2 ;
 - 3: $CAG_1 \leftarrow$ CSOA(AG_1, IG_1);
 - 4: $AG_{NEW} \leftarrow$ `update_access_graph`(CAG_1, AG_2);
 - 5: `resolve_conflicts`(AG_{NEW}, IG_2);
 - 6: $CAG_2 \leftarrow$ CSOA(AG_{NEW}, IG_2);
 - 7: Return offset assignment based on CAG_2 ;
-

Function `resolve_conflicts()`. Two variables that were coalesced in the old assignment may interfere with each other after the code update. ICSOA identifies this as a *conflict* and uses the function `resolve_conflicts()` to resolve it.

The function first orders the variables in each coalescing group, by the factor

$$\frac{\text{Num}_{\text{local_itfs}}}{\text{Num}_{\text{local_acs}}}$$

where $\text{Num}_{\text{local_itfs}}$ represents the number of interferences between this variable and the other group members, and $\text{Num}_{\text{local_acs}}$ represents the number of adjacent accesses between this variable and the other group members. The function then extracts the interfering variable that has the highest factor value from the group. After several iterations, all the internal conflicts within one group will be resolved. By doing so, the variables that create more interference but have fewer adjacent accesses with the others are extracted earlier from the coalescing group.

For each variable chosen to be extracted from the coalescing group, the function splits the live range (i.e., conflict range) into two subranges, the original part and the newly extended part. We use the old variable name to represent the original subrange, and introduce a *patch variable* for the extended subrange. To ensure semantic correctness, we insert $a'=a$ or $a=a'$ to move the value between the subranges. The insertion involves memory copy and tends to incur large overhead. We will evaluate its impact in the experiments.

For the example in Figure 23, ICSOA combines the coalesced offset assignment (Figure 23(d)) and the new access graph (Figure 24(c)). Figure 27(a) shows the updated access graph. As there is no conflict between the access graph and interference graph (Figure 24(b)), ICSOA outputs the same coalesced assignment (Figure 27(c)). In this example, the script generated from ICSOA is 71% smaller than that of recompilation using CSOA.

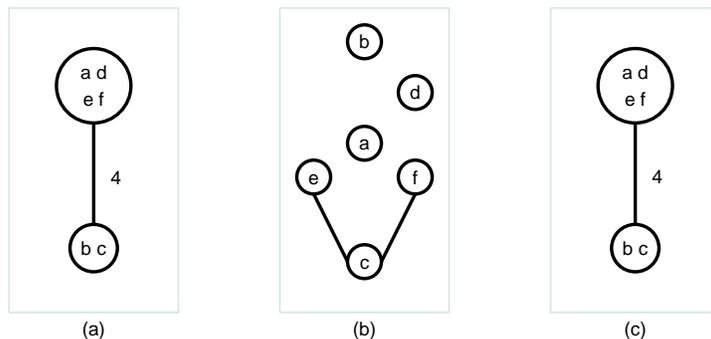


Figure 27: An example of ICSOA scheme: (a) AG_{NEW} , the updated access graph; (b) IG_2 , the new interference graph; (c) The final offset assignment.

3.3.3 Incremental coalescing general offset assignment (ICGOA)

In practice, multiple address registers are available on DSP chips. In this section, I extend the update-conscious ICSOA design to handle multiple address registers, that is, to design an incremental coalescing GOA scheme for general offset assignment.

Compared to SOA, GOA has one more task, i.e., the address register allocation problem — which variable should use which address register at each program point. Existing heuristics [51, 86] have found that it is beneficial to let each address register handle a subset of

variables. After code updates, there exist two options. One is to repartition the variables to be handled by different address registers; the other is to keep the variable partition handled for each address register. Repartitioning variables greatly affects the addressing mode of all instructions that access these variables — my experimental data showed that it always generates high-level updates. Therefore, ICGOA tries to keep the variable partition the same as the old binary when the update is at a small or medium level, and repartition variables when the update is at a high level.

Similar to ICSOA, ICGOA is developed based on CGOA [59, 86]. The detailed algorithm is presented in Algorithm 3. It first divides the variables into several groups such that the variables in the same group share the same address register; within each group, ICSOA algorithm proposed above is applied to find the layout of each group; the complete memory layout is the combination of the results of all groups.

Algorithm 3 Incremental coalescing-based GOA (ICGOA).

Input: AS_1, AS_2 : access sequences before and after update;

IG_1, IG_2 : interference graphs before and after update;

N_{AR} : the number of address registers;

Output: the offset assignment.

{Run CGOA over the original code}

1: $Partition_1[N_{AR}] \leftarrow CGOA(AS_1, IG_1, N_{AR});$

{Remove the deleted variables and partition the newly added variables}

2: $Partition_2[N_{AR}] \leftarrow ICGOA_Partition(Partition_1[], N_{AR}, AS_2, IG_2);$

{Run ICSOA in each variable partition group}

3: **for** $i = 0$ to N_{AR} **do**

4: $Offset[i] \leftarrow ICSOA(Partition_2[i], AS_1, AS_2, IG_1, IG_2);$

5: **end for**

6: Return $Offset$;

The algorithm first produces the variable partition of the old version based on the old access graph and interference graph, using a heuristic-based algorithm [59]. In this algorithm, the variables are sorted by the decreasing order of the *global interference number* (Num_{global_itfs}), which is the total number of interferences that each variable has with the other variables. The variable that has the highest *global interference* should be partitioned first, because it has the most constraints. Then CGOA determines the partition group that the variable belongs to, according to the *local interference numbers* (Num_{local_itfs}). This number represents the number of interferences that this variable has with the other variables

within a partition group. The variable is assigned to the group that has the smallest number of *local interferences* with it. This partition result is saved in $Partition_1$ in algorithm 3.

Based on the old partition result $Partition_1$, the removed variables are first deleted from each partition. Similar to CGOA, my algorithm first orders the new variables according to their *global interference numbers* and then iteratively assigns each variable to the group that has the fewest *local interferences*. The generated new partition $Partition_2$ is similar to the old one, as it only incorporates the variable changes to the old partition. After that, ICSOA is applied within each partition group to generate the memory layout for the variables inside each group. The global memory layout is then the results of all groups.

4.0 SOFTWARE DIFFERENTIAL PATCHING

As shown in in Figure 28, after generating the new binary E' using UCC, the framework summarizes the code difference between E' and the old binary E in a patch script P , and then disseminates it to remote sensors. After receiving the patch, each sensor then reconstructs the new binary image E' by combining P with the old binary image E that already exists on the sensor.

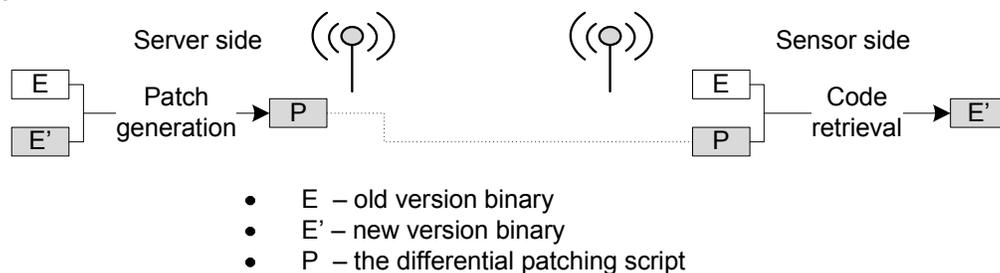


Figure 28: Patch generation and binary reconstruction.

The design of the script primitives affects both the update data packet transmission effectiveness and the runtime overhead on each sensor node. To facilitate the description of UCC techniques, I divide the binary level changes into two categories – *instruction-based changes* and *data-based changes*. Different script formats are used to describe different changes. I adopted four simple *instruction-based* primitives from the prior work [68], and proposed three advanced *instruction-based* primitives and three *data-based* primitives to describe the higher-level code changes.

Each remote sensor has a patch interpreter that can interpret the primitives in the patch and construct the new binary accordingly. As the number of primitive types increases, the patch interpreter shares some similarity with a JIT (just-in-time) compiler: (i) both convert

a small file to the executable binary that can run on the target processor; (ii) both need additional files in the translation. A JIT compiler needs additional library files while a patch interpreter needs the old binary. However, there are differences: (i) the old binary can be removed after generating the new one, while the libraries used by a JIT compiler have to be kept for future use. Keeping the libraries on sensors wastes precious storage resources, as the libraries are only needed at the time of generating the new binary, and most functions in the libraries are rarely used; (ii) a JIT compiler is also much more complicated and does many jobs that a patch interpreter does not need to do. For example, a JIT compiler needs to perform pointer analysis in order to disambiguate memory accesses [73]. The time and space overhead requirements of a JIT compiler are much higher than a patch interpreter.

4.1 INSTRUCTION-BASED PATCHING

I use the instruction-based primitives to describe the code changes such as adding, removing, or updating instructions at the binary level. Figure 29 lists the format of these script primitives. The simple primitives are adopted from the prior work [68], while the advanced primitives are developed to solve more complicated code update cases. The difference between these two types is that advanced primitives are not used to describe simple bit-level differences, but rather high-level structure changes.

4.1.1 Simple primitives

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

Figure 30 shows an example of the simple primitives. The new version contains three chunks of code, $[100,110]$, $[112,130]$, and $[132,140]$. While the first chunk and the third chunk can be found in the old code, the second chunk is new. Therefore, the update script

Primitive	Format and Operation	Size (bytes)
insert	<div style="display: flex; align-items: center;"> <div style="border: 1px solid black; padding: 2px; margin-right: 10px;">0000 xxxx</div> <div style="border: 1px solid black; padding: 2px; margin-right: 10px;">data</div> <div style="margin-right: 10px;">...</div> <div style="border: 1px solid black; padding: 2px;">data</div> </div> <p style="text-align: center;">number of bytes to be inserted</p>	1 + number
replace	<div style="display: flex; align-items: center;"> <div style="border: 1px solid black; padding: 2px; margin-right: 10px;">0001 xxxx</div> <div style="border: 1px solid black; padding: 2px; margin-right: 10px;">data</div> <div style="margin-right: 10px;">...</div> <div style="border: 1px solid black; padding: 2px;">data</div> </div> <p style="text-align: center;">number of bytes to be replaced</p>	1 + number
copy	<div style="border: 1px solid black; padding: 2px; margin-bottom: 5px;">0010 xxxx</div> <p style="text-align: center;">number of bytes to be copied</p>	1
remove	<div style="border: 1px solid black; padding: 2px; margin-bottom: 5px;">0011 xxxx</div> <p style="text-align: center;">number of bytes to be removed</p>	1
shift	<div style="display: flex; align-items: center;"> <div style="border: 1px solid black; padding: 2px; margin-right: 10px;">0100 0000</div> <div style="border: 1px solid black; width: 40px; height: 15px; margin-right: 10px;"></div> <div style="border: 1px solid black; width: 40px; height: 15px;"></div> </div> <p style="text-align: center;">[A1, A2, S] 3 words: shift S bytes from A1 to A2</p>	7
clone	<div style="display: flex; align-items: center; margin-bottom: 5px;"> <div style="border: 1px solid black; padding: 2px; margin-right: 5px;">0101 0000</div> <div style="border: 1px solid black; padding: 2px; margin-right: 5px; font-size: 8px;">start_addr</div> <div style="border: 1px solid black; padding: 2px; margin-right: 5px; font-size: 8px;">end_addr</div> <div style="border: 1px solid black; padding: 2px; margin-left: 10px; font-size: 8px;">register mappings</div> </div> <p style="text-align: center;">2 words 1 word * number of register mappings</p> <p style="text-align: center;">[A1,A2,Ra1...Ra32,Rb1...Rb32] clone instructions [A1,A2]. Replace all Rai with Rbi.</p>	5 + 2*number
insert_access	<div style="display: flex; align-items: center;"> <div style="border: 1px solid black; padding: 2px; margin-right: 10px;">0110 xxxx</div> <div style="border: 1px solid black; padding: 2px; margin-right: 10px;">data</div> <div style="margin-right: 10px;">...</div> <div style="border: 1px solid black; padding: 2px;">data</div> </div> <p style="text-align: center;">number of bytes to be inserted</p>	1+number

Figure 29: The instruction-based patch script primitives

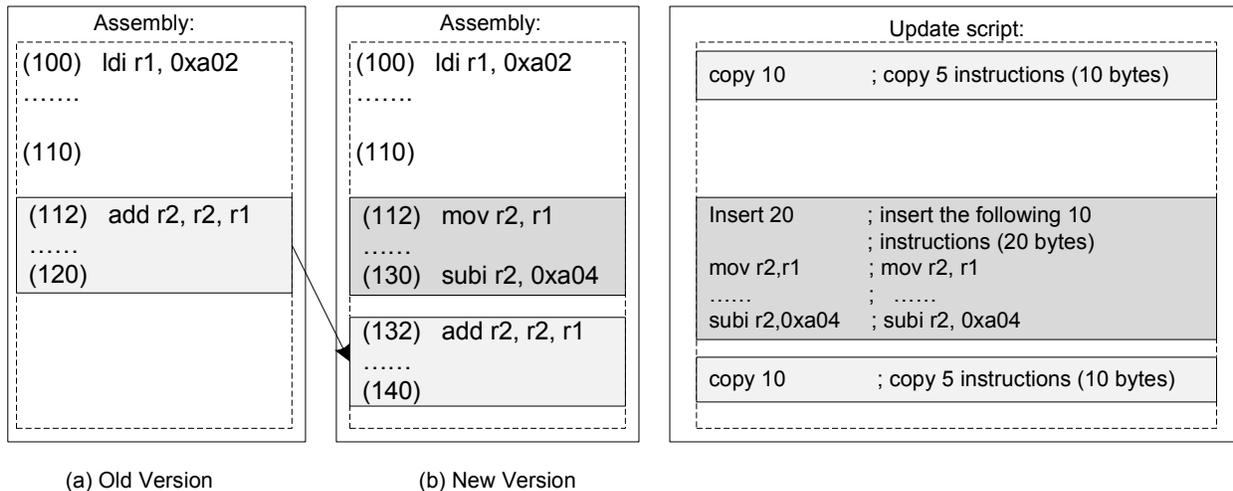


Figure 30: An example of the simple primitives. New code [112,130] is inserted. [112,120] in the original code is now moved to [130,140] in the new version.

contains two `copy` primitives and one `insert` primitive. The `insert` primitive has a one-byte opcode and 10 instructions (or 20 bytes). The total size of the update script is 23 bytes.

In order to interpret the simple primitives on remote sensors, the script interpreter maintains two instruction pointers; one points to the old binary image and the other points to the last instruction that was just re-generated in the new binary image. Note that as the new binary is being generated, the *last* instruction at this point is not the real last instruction of the new binary. The `insert` primitive inserts the instructions in its data part into the new binary image, and moves the pointer pointing to the new code to the end (of the partially generated binary). The `replace` primitive does the same thing to the new binary but also moves the pointer in the old binary for the same distance. The `copy` primitive copies instructions from the old binary to the new one, and moves both pointers.

4.1.2 Advanced primitives

In my experiments, I observed that many changes are caused by the same high-level update. Just using the simple primitives may not result in a compact update script. There are demands to design fancy primitives that can present more than one change. Based on this observation, I propose three advanced primitives in my design.

4.1.2.1 The `shift` primitive

Inserting or removing some code from the old binary affects both the absolute addresses of the code below the affected segment, and the distance between the code above and below the segment. As a result, many branch instructions may need to be updated. Instead of summarizing these changes in multiple simple primitives, it is more energy-efficient to use a new `shift` primitive [68] to inform the remote sensors that a segment has shifted. I adopted this primitive in my experiments and found that it can significantly reduce the script size.

As Figure 29 shows, the `shift` primitive contains a one-byte opcode and another three words to indicate that the code segment $[A1, A2]$ is now moved to $[A1+S, A2+S]$. All the branch/jump instructions whose destination addresses are within the range $[A1, A2]$ should have the destination addresses shifted by S .

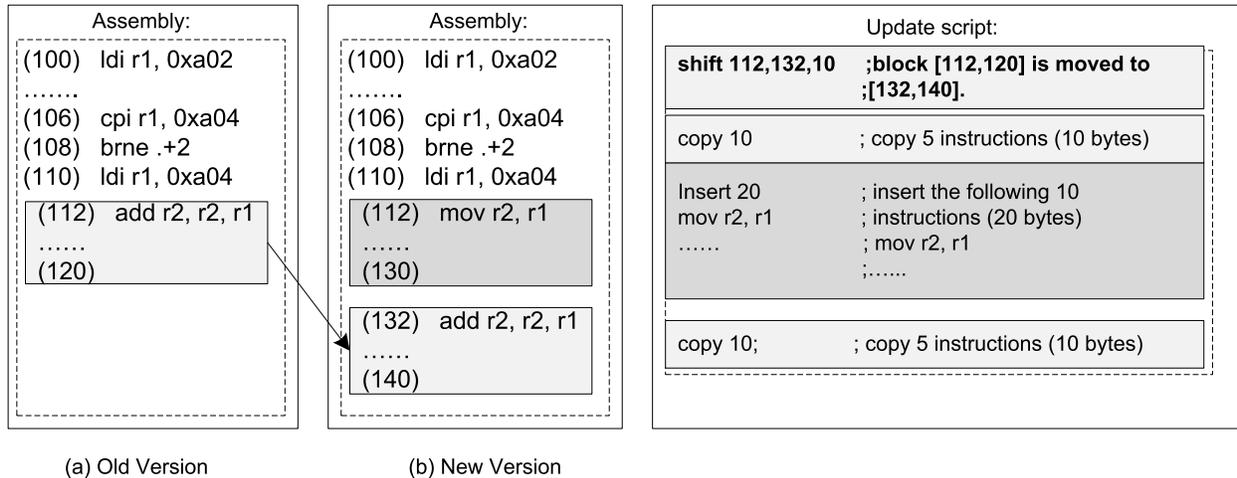


Figure 31: An example of the `shift` primitive. New code $[112,130]$ is inserted. $[112,120]$ in the old code is now moved to $[132,140]$ in the new version. Some control flow instructions are affected due to this address change.

Figure 31 shows an example of 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. All the branch instructions that jump to any instruction inside 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 the `shift` primitive helps to reduce the script size. The tradeoff is that a slightly more powerful interpreter has to be installed at the sensor side — it needs to decode each instruction type to extract the bits indicating the desired target address.

The `shift` primitive is implemented as follows. A shift information table is maintained at the sensor side. When encountering a `shift` primitive, the interpreter adds into the table one shift entry that includes the start address, end address, and shift offset. When copying one instruction from the old binary, the interpreter checks if it is a branch/jump instruction. For the branch instructions that use absolute addresses, the interpreter determines if the original destination falls in the shift range, and if yes, it updates the address. For the branch

instructions that use relative addresses, the interpreter needs to first compute the target address by adding the relative offset to the address of the current instruction.

4.1.2.2 The `clone` primitive

When inlining the same function at different places in the code, the binary code segments often look similar but use different sets of registers in different contexts. This is because different free registers are available at different places. Unfortunately, when an `inline` function gets changed, all these code segments need to be updated, which creates redundancy that can be optimized.

Based on this observation, I introduce the `clone` primitive. When an `inline` function is inserted or modified in the code update, the update script only includes one copy of the binary generated by the `inline` function, and advises the sensors to replicate the master copy with register usage replacement while constructing the binary for the other instances of this `inline` function.

The example below shows how the `clone` primitive works. Assume that an `inline` function is called at multiple locations, such as blocks $[A1, A2]$ and $[B1, B2]$. The patch generator uses block $[A1, A2]$ as the base, and tries to map the register allocation between these two blocks. Assume the register mapping between them is shown below, $(R_{a1}, R_{a2}, \dots, R_{an}) \Rightarrow (R'_{b1}, R'_{b2}, \dots, R'_{bn})$. Given such information, instead of using a sequence of the simple primitives to describe the updated/new code of block $[B1, B2]$, my scheme copies instructions from block $[A1, A2]$, and slightly changes the register assignments according to the register mapping to rebuild block $[B1, B2]$.

As shown in Figure 29, the `clone` primitive has one-byte opcode, another several bytes to specify the starting and ending addresses of the code segment where the code would be copied from, and the register mappings. The primitive length varies according to the register mapping complexity. Assume there are N pairs of register mappings, the `clone` primitive length is $5+2*N$. An `inline` function may have multiple instances in the binary image. Since the new binary is re-generated in increasing order, the instance that is stored with the lowest addresses is chosen to be the master copy, while all other instances clone the code from the

master copy.

Figure 32 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. It specifies that the second code block clones the block $[100,106]$ while registers r_1 and r_2 need to be updated to be r_{11} and r_{12} respectively.

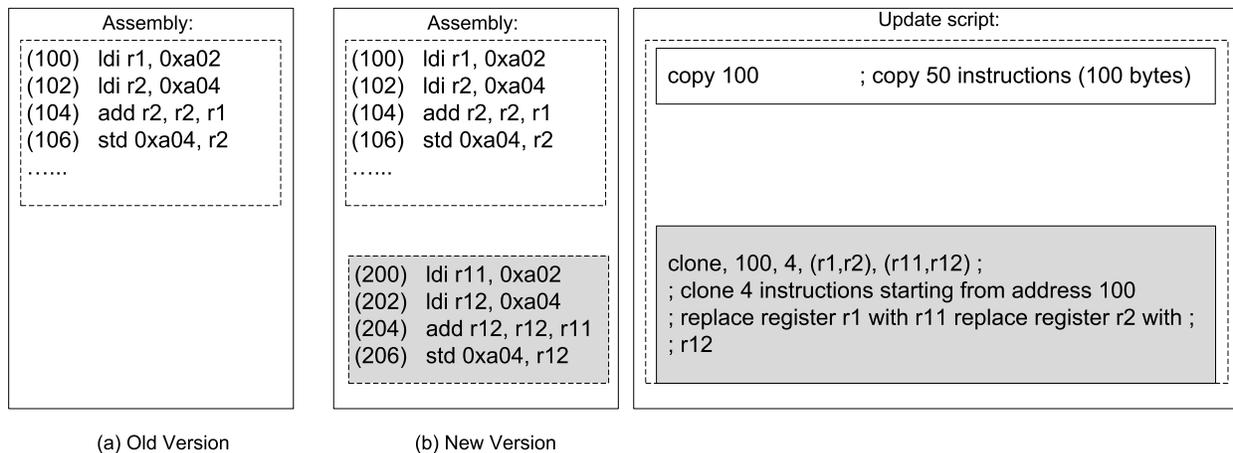


Figure 32: An example of the `clone` primitive. New code $[200,206]$ is inserted, which is compiled from the same `inline` function as code $[100,106]$.

The `clone` primitive can reduce the script size when the `inline` function is called more than once, and the register mapping can be easily created. However, if the register mapping is too complicated, the script size could be very big such that it is better to use the simple primitives, such as `insert` and `replace`. In addition, using the `clone` primitive requires the sensor-side interpreter to decode different instruction types to replace register names. Each clone operation needs to fetch the master copy, decode the instructions, and replace all mapped registers.

The `clone` primitive is similar to the echo instruction proposed for code compression. The basic echo instruction proposed by Fraser [25] only supports compressing exactly the same code sequence [25]. Lau *et al.* [43] proposed `bitmask echo` to compress a slightly different code sequence. They used a bitmask that only supports skipping some instructions in a code sequence. For the sequences that use different sets of registers (the above example), they

proposed to insert `mov` instructions, which increases the runtime overhead. In contrast, the `clone` primitive is used to generate the binary and does not hurt the runtime performance.

4.1.2.3 The `insert_access` primitive

For DSP applications, when inserting a new memory access between two existing accesses, my patch generator needs two `replace` primitives and one `insert` primitive, as shown in Figure 33(d). Since the update primitives only modify the addressing modes, a compact way to express this update is to include the memory address difference in the script and let the remote sensor generate the correct addressing modes for the related instructions. Thus, I introduce an advanced primitive – `insert_access`.

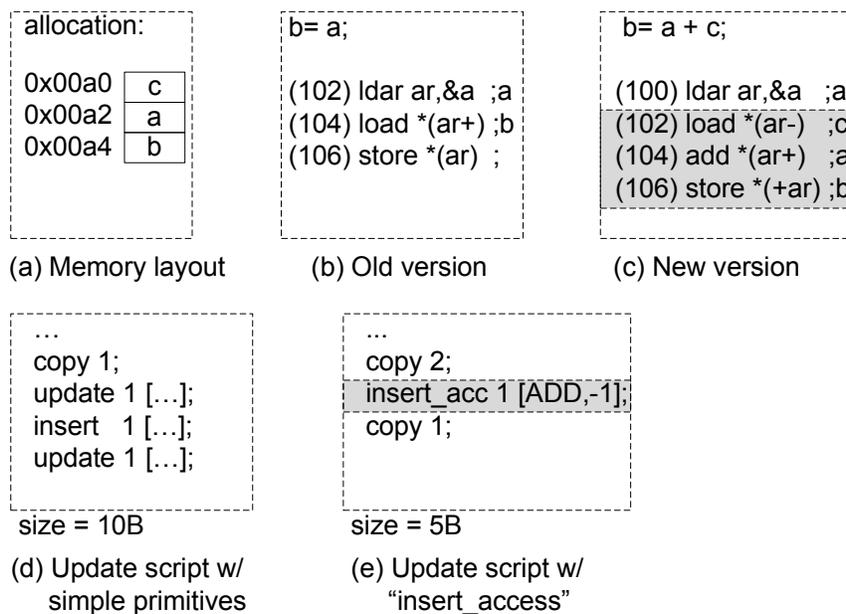


Figure 33: An example of the `insert_access` primitive: (a) The memory layout for both versions; (b) The source and assembly before code update; (c) The source and assembly after code update; (d) The update script using the simple primitives; (e) The update script using the `insert_access` primitives.

The `insert_access` primitive is similar to the `insert` primitive, except that its data field is specified as follows:

[operation, δ_{diff}]

where δ_{diff} represents the address difference between the locations accessed by the current instruction and the preceding instruction respectively. In the example (Figure 33(c)), the new access is `c` (located in memory slot 0), and the preceding memory access is `a` (located in memory slot 1), so δ_{diff} is -1. Since it is the `add` operation that accesses `c` in the new instruction, the update primitive is

```
insert_access 1 [ADD, -1].
```

Rewriting the update script of the example using the `insert_access` primitive, the script size is reduced by 50% (Figure 33(e)).

The `insert_access` primitive allows the sensors to correct the addressing modes before and after the newly inserted memory access using the following method. Let us call the last memory access instruction before the inserted instruction the *predecessor* and the first one that is executed after the inserted instruction the *successor*. The offset between them can be calculated. The `insert_access` primitive encodes the offset between the inserted memory access and the *predecessor*, thus the offset between each two instructions among these three can be calculated. Based on that information, the addressing modes can be determined.

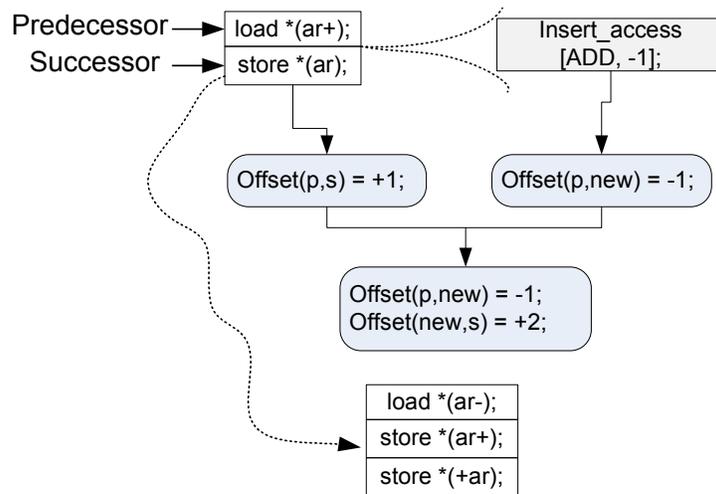


Figure 34: An example of the interpretation procedure of the `insert_access` primitive.

Figure 34 demonstrates the interpretation procedure of the example in Figure 33. Knowing the offset between the *predecessor* and the inserted instruction is -1, and the offset between

the inserted instruction and the *successor* is +2, the addressing mode of the *predecessor* can be determined to be pre-incremental (as discussed in Section 2.2.2), that of the inserted instruction can be determined to be post-incremental, and that of the *successor* can be determined to be pre-incremental.

4.1.3 Sensor-side primitive interpretation

At the sensor side, the received patch is first stored in the flash memory. When the script download is complete, each sensor then runs a simple script interpreter to incrementally reconstruct the new binary image. The reconstruction is based on the received patch and the old binary image that resides in the program memory. The generated new binary image will be first stored in the flash. When the primitive interpretation is complete, the sensor loads the new binary image back to the program memory and restarts to execute the new version. When distributing the patch, message authentication code (MAC) using authentication protocols such as μ TELSA [62] may need to be added to ensure the data integrity of patch packets. That is, only the patch distributed from the trusted sink node may be used to re-generate the new binary and substitute the old binary.

The program memory supports random access such that the pointer pointing to the old binary can be moved freely. However, one limitation of the flash memory is that it has to be programmed at block levels, e.g., 256 bytes on Mica2 sensors [19]. In order to change one byte, the sensor has to read the corresponding block into data memory, modify it, and then write it back. Thus, the constructed new binary needs to be buffered in the data memory until it reaches the size of a flash block, then the code block will be written back to the flash. The size of the temporary code buffer should be a multiplier of the block size.

The interpretation algorithm is presented in Algorithm 4. The interpreter maintains two instruction pointers: one points to the old binary image, and the other points to the last instruction that was generated in the (partial) new binary image. Each script primitive is scanned once, and is interpreted to construct the new binary.

For the `insert` and `replace` primitives, the interpreter copies the instructions from the data part of each primitive to the new binary. For the `copy` primitives, the interpreter copies

Algorithm 4 Primitive interpretation and code reconstruction.

Input: Pointer to the beginning of the patch script P_S ,

Pointer to the beginning of the old binary P_O ,

Pointer to the beginning of the new binary P_N .

```
1: for (;  $P_S \neq \text{script.end}()$ ;  $P_S = \text{script.next\_primitive}()$ ) do
2:   switch(primitive_type( $P_S$ ))
3:     case insert:
4:       write_code_buffer( $P_N$ , insert_data( $P_S$ ), insert_bytes( $P_S$ ))
5:       break
6:     case replace:
7:       write_code_buffer( $P_N$ , replace_data( $P_S$ ), replace_bytes( $P_S$ ))
8:        $P_O += \text{replace\_bytes}(P_S)$ 
9:       break
10:    case copy:
11:      write_code_buffer( $P_N$ ,  $P_O$ , copy_bytes( $P_S$ ))
12:       $P_O += \text{copy\_bytes}(P_S)$ 
13:      break
14:    case remove:
15:       $P_O += \text{remove\_bytes}(P_S)$ 
16:      break
17:    case shift:
18:      add [  $A1(P_S)$ ,  $A2(P_S)$ ,  $S(P_S)$  ] to addr_shift_table
19:      break
20:    case clone:
21:      if ([start_addr( $P_S$ ), end_addr( $P_S$ )] is not in clone_buffer)
22:        load code [start_addr( $P_S$ ), end_addr( $P_S$ )]  $\Rightarrow$  clone_buffer;
23:      endif
24:      replace_register(buffer, register_pairs( $P_S$ ))
25:      write_code_buffer( $P_N$ , buffer, clone_bytes( $P_S$ ))
26:      break
27:    case insert_access:
28:      update the addressing mode of  $P_N-1$  if necessary
29:      generate the addressing mode addr_mode for the inserted instruction
30:      instruction  $i = \text{form\_inst}(\text{opcode}(P_S), \text{addr\_mode})$ 
31:      write_code_buffer( $P_N$ ,  $i$ , length( $i$ ))
32:      break
33:    default:
34:      error("no such primitive")
35:   end switch
36: end for
37: copy new_binary to program_memory
38: restart the sensor
```

the instructions from the old binary image instead. The two pointers are updated as well. The pointer in the new binary always points to the end of the partially generated image. The pointer in the old binary is shifted according to the number of bytes that have been copied, removed, or updated.

For the `clone` primitives, the interpreter reads the master copy from the program memory and modifies the register names to construct the cloned copy. To save the time and energy to read the master copy multiple times, the interpreter buffers it in the data memory, provided there is enough space. For the `insert_access` primitives, the interpreter needs to decode the *predecessor* and *successor* to correct their addressing mode. In the worst case, the predecessor may have been written into the flash. The interpreter then needs to reload the block and modify, which is very inefficient. Since `insert_access` only inserts one or two instructions, the interpreter avoids reloading by prefetching — when the code buffer is full and needs to flush, the interpreter prefetches the next primitive to ensure that if the last instruction in the code buffer is a predecessor, its addressing mode gets updated before flushing to the flash memory.

When encountering a `shift` primitive, the interpreter adds one entry that records the start address, end address, and shift offset into the shift table. The algorithm shown in Algorithm 5 is then called whenever an instruction is constructed and written to the temporary code buffer. A simple decode operation is done first to filter out the branch instructions. If the target address of the branch instruction falls in any range that needs address shifting, this instruction gets updated for the address shift. When the temporary buffer is full, the generated code gets flushed into the flash memory.

The memory space required for the interpreter includes the temporary code buffer and the shift table. As discussed before, the minimal size of the code buffer is the block size of the program flash, which is 256 bytes for Mica2 sensor. Each entry of the shift table includes the start address, end address, and the shift offset. As the program memory size is 4KB, the start address and end address can be encoded using 3 bytes. The shift offset can be encoded using 1 byte. Thus, the storage required for each entry is 4 bytes.

Algorithm 5 write_code_buffer /*write the constructed code into code buffer*/

Input: Destination address & P_N ,Source address P_O ,Number of bytes to be copied *nbytes*.

```
1: memcpy( $P_N, P_O, nbytes$ );
2: for all instructions i to be copied do
3:   if inst_type(i) == branch/jump then
4:     target = target_addr( $P_N$ )
5:     if there exists a shift entry  $e \in$  shift_table, where  $target \in [e.A1, e.A2]$  then
6:       target = target + e.offset
7:       change the target address of  $P_N$  to target
8:     end if
9:   end if
10: end for
11:  $P_N += nbytes$ 
12: if  $P_N == code\_buffer.end$  then
13:   write_to_flash(code_buffer)
14:    $P_N = code\_buffer.begin$ 
15: end if
```

4.2 DATA-BASED PATCHING

Sometimes, binary level changes at several places may be caused by one memory layout change. While these changes still manifest as instruction differences, the root cause is data allocation change. For example, if a relocated variable **a** is loaded or stored in several places, a simple script may need multiple primitives, each of which indicates one instruction level change. Instead, if the script interpreter at the sensor side can decode instructions and identify all loads and stores of **a**, it is possible to send one “relocate **a**” primitive to minimize the update script size.

Identifying relocated variables is particularly beneficial for DSP applications due to the close connection of data allocation and instruction format (addressing mode). In this section, I focus mainly on DSP applications and refer to the binary instructions that are inserted, removed, or changed due to the offset assignment changes as *addressing mode change* (AMC) instructions. The design goal of introducing *data-based primitives* is to reduce the transmission of AMC instructions, and let the remote sensor construct these instructions. Compared to instruction-based primitives, one data-based primitive updates the code in several places.

4.2.1 Data-based primitives

Figure 35 lists the *data-related* primitives that are used to specify the memory layout change. I only consider the allocation of scalar variables. Each memory location contains one variable or multiple coalesced variables [59, 86, 87]. Each sensor stores the old memory layout, and after receiving the layout change primitives, it first reconstructs the new memory layout and then updates the instructions. Both the old and new memory layouts are maintained as tables. Each entry contains the variables that share the same memory slot, and all entries are ordered by their addresses.

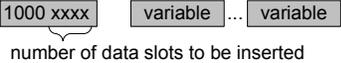
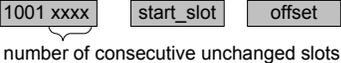
Primitive	Format and Operation	Size (bytes)
copy_slot	 number of data slots to be copied	1
insert_slot	 number of data slots to be inserted	1+number
shift_slot	 number of consecutive unchanged slots	3

Figure 35: The primitives to patch the data allocation.

The `copy_slot` primitive copies multiple memory slots from the old memory layout to construct the new memory layout. This is similar to the `copy` primitive to update instructions. The `insert_var` primitive adds a list of variables to the active memory slot of the new memory layout table. The insertion can be caused by adding a new variable, or by moving an existing variable from another location. The latter implicitly has the variable removed from the old slot, which is omitted to keep the script compact. The `shift_slot` primitive represents the case that multiple slots may be grouped and shifted from the old memory location to the new memory location. The `shift_slot` primitive specifies the number of slots to be shifted, the starting point of the shift, and the shift offset.

4.2.2 Sensor-side primitive interpretation

After receiving the update script, each sensor interprets the *data-based* primitives to generate the new memory layout, and then interprets the *instruction-based* primitives to construct

basic blocks by inserting, removing, or updating certain instructions of the old binary. The interpreter fixes the addressing mode of each instruction in a basic block according to the new memory layout, and then writes the completed block into the flash. There are two pointers pointing to the memory slot in the new and old memory layout table respectively to assist the interpretation of data-based primitives.

Figure 36 illustrates the update procedure. ICSOA coalesces multiple variables, both **a** and **e**, in one memory location **0x00a2**, and an update may relocate **e** to **0x00a0** while keeping **a** in the same memory slot. This complicates the sensor-side update as some accesses to **0x00a0** should be updated while others should not. Clearly, additional information is required to fix the addressing modes at the sensor side.

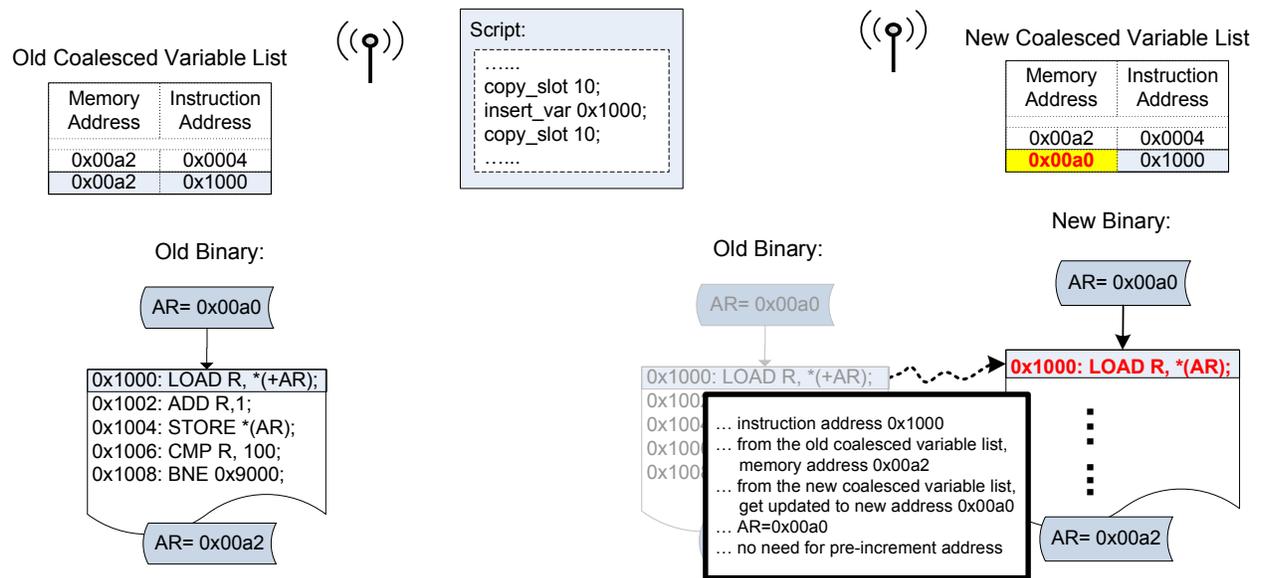


Figure 36: The code construction procedure of the data-based primitives. The left shows the server side, and the right shows the sensor side update).

To solve this problem. I use an implicit pointer to track the current memory slot when copying from the old layout to the new layout. “insert_var 0x1000” inserts **e** into the current slot, i.e. **0x00a0**. Here variable **e** is represented using its instruction address **0x1000**. A record can be found in the coalesced variable list indicating this mapping, and will be updated to reflect the re-allocation.

To update its addressing mode in the new code, a query is sent to the coalesced variable list, from which the interpreter knows that this instruction accesses 0x00a0 instead of 0x00a2. Since AR contains 0x00a0 when entering the basic block, there is no need for pre-increment. Similar decisions are made for other instructions in the basic block. The interpreter also needs to ensure the exiting AR contains 0x00a2.

Therefore, the interpreter needs the following information to fix the addressing modes:

- A coalesced variable list to distinguish each of the coalesced variables; and
- The AR values when entering and exiting each basic block.

4.2.2.1 Auxiliary data structures

To correctly update the code with a memory layout change, e.g., when `a` is relocated, the interpreter needs to locate all of `a`'s uses and ensure the AR contains the correct address when accessing `a`. Conceptually, this can be done by a relocation table. Unfortunately, a traditional relocation table identifies all the places that the binary code accesses the memory. Since DSP code relies heavily on offset assignment and accesses the memory frequently, adopting a traditional relocation table would generate a table linear to the size of the binary code. Instead, I introduce the following two lightweight auxiliary data structures to enable relocatable DSP code.

Coalesced variable list. The coalesced variable list is designed to differentiate the coalesced variables in one memory location. If a memory location contains only one variable, then the scheme does not allocate any entry in the list. If multiple variables are coalesced and stored in the same memory location, the scheme allocates the entries as follows.

Memory Address	Instruction Address
0x00a2	0x0004
0x00a2	0x1000

Figure 37: Coalesced variable list.

Since the coalesced variables have their accesses spread in the code, I group consecutive definitions/uses that access the same variable and allocate one entry to each group. This is done based on the code text without considering the control flow, or the variable live ranges, etc. For example, if the live ranges of two coalesced variables overlap due to linear layout of control structures such as branches, then we allocate one entry for each segment. As shown in Figure 37, each entry contains two fields: the memory slot address, and the starting instruction address of each code text segment.

For example, variables `a` and `e` share the same memory location `0x00a2`. The live ranges of `a` and `e` are `[0x0000,0x0004]` and `[0x0010,0x1000]` respectively. Figure 37 illustrates the coalesced variable list. Given a memory access to `0x00a2`, we can easily differentiate whether it is accessing `a` or `e`.

The original coalesced variable list is preloaded on the sensors before deployment. The updates to the coalesced variable list are transmitted with the code update script.

AR in/out value list. To generate the correct addressing mode at the sensor side, the AR in and out values for each basic block are also needed. I choose to construct the list rather than building the control flow graph on demand to reduce the memory and complexity overheads. This table contains the starting and ending addresses, the address register's entering and exiting values, and the successive basic block(s) of each basic block, as shown in Figure 38.

Index	Starting Address	Ending Address	AR In	AR Out	Successive Basic Blocks
10	0x1000	0x1008	0x00a0	0x00a2	20

Figure 38: The AR in/out value list.

The original list is preloaded on the sensors before deployment. The interpreter automatically generates the new list while generating the new binary code.

The AR out value of a basic block may affect the addressing mode of its successive basic blocks. The situation becomes more complicated if there are multiple predecessors (or successors). Synchronization needs to be done among these predecessors (or successors),

which may cascadingly affect other instructions in those basic blocks. To simplify the code update on the sensor side, the server explicitly sends out the AMC instructions that follow an inserted/updated/removed instruction when it is the last instruction of a basic block.

Complexity analysis. Algorithm 6 presents the detailed steps that are used to correct the addressing modes due to data-based primitives. The extra interpretation overhead is to look up the address register value for the first instruction of each basic block, keep track of this value while constructing the instructions in the basic block, and generate the correct addressing mode for each memory access instruction. However, addressing mode correction is only necessary when the data layout is changed for the corresponding code segment. For example, if the highlighted variable list change is the only memory layout change in the example shown in Figure 36, the instructions before 0x1000 do not need to be decoded because the memory layout change does not affect those instructions.

Each entry of the “coalesced variable list” is 4 bytes, and each entry of the “AR in/out value list” is 9 bytes. For code and updates that are of small to medium sizes, the complete two lists are placed in the data memory for fast access. If the code and/or the update is large, the lists may be too large to fit in the memory. In this case, the lists are stored in the flash while a subset resides in the memory. The patch is divided into several sub-patches while each sub-patch only needs a subset of the complete lists that can fit in the memory. At the beginning of each sub-patch, explicit primitives are needed to set up the in-memory subsets. For this purpose I introduce a new primitive type — `load LstT, EntryRange`, where `LstT` indicates one of the two lists; and `EntryRange` is a range indicating the list entries that should be loaded from the flash and reside in the memory. For example, `load 0, [10,20]` indicates the entries from 10 to 20 of the coalesced variable list should be loaded into the memory. I did not include this primitive in the experiment section as the lists can always fit in the memory for the test cases I studied.

Algorithm 6 *addr_mode_correction* /*Correct the addressing mode of an instruction*/

Input: Instruction *i* which will be copied from the old binary to the new binary,
address of this instruction in the old binary **addr1**,
address of this instruction in the new binary **addr2**

Output: Instruction *i'* which has the same opcode as *i* and with the addressing mode corrected

```
1: if inst_type(i) is memory access instruction then
2:   {find out the value stored in address register (AR)}
3:   if i is the first instruction of basic block B1 in old binary then
4:     old_ar_value = query_old_AR_tab(B1.AR_in)
5:   end if
6:   if i is the first instruction of basic block B2 in new binary then
7:     new_ar_value = query_new_AR_tab(B2.AR_in)
8:   end if
9:
10:  {find out the memory address that this instruction tries to access}
11:  old_mem_addr = gen_addr_mode(old_ar_value, addr_mode(i))
12:  {find out the variable that this instruction tries to access}
13:  var_name = query_old_var_tab(old_mem_addr, addr1)
14:  {find out the new memory location of this variable}
15:  new_mem_addr = query_new_var_tab(var_name, addr2)
16:
17:  {generate the new addressing mode and construct the instruction}
18:  addr_mode = form_addr_mode(new_mem_addr, new_ar_value)
19:  instruction i' = form_inst(opcode(i), addr_mode)
20:  return i'
21: end if
```

5.0 DISTRIBUTION PROTOCOL

After minimizing the binary code difference using update-conscious compilation and summarizing the difference in a patch script using simple and advanced primitives, the sink node is ready to disseminate the patch to remote sensors.

In Chapter 1, I have discussed that there are two types of update requests — software upgrade and software switch. Software upgrade refers to the problem of updating the code of all sensors; software switch refers to the problem of updating the code of a subset of sensors in a MA-WSN. For software upgrade, only the sink node is the source node before the update; while for software switches, both the sink node and many sensors can be the source nodes. Several code dissemination protocols have been proposed for software upgrade [32, 41, 49] and for software switch [83].

In this chapter, I will first discuss the Deluge [32] and Melete [83] protocols. I will then present my multicast-based code redistribution protocol (MCP) for achieving energy-efficient code dissemination.

5.1 BROADCAST-BASED CODE DISTRIBUTION PROTOCOLS

5.1.1 Deluge: an effective code dissemination protocol for SA-WSNs

Deluge [32], the default code dissemination protocol in TinyOS, supports code dissemination in a multi-hop SA-WSN (single application wireless sensor network). Deluge first updates the nodes around the sink node, and then gradually updates the nodes that are one hop away from the updated nodes. It divides the code to be distributed into several pages, and each page consists of multiple packets. To adapt to the lossy wireless communication

environment, packets within one page may be received out of order. Received packets are first buffered in the data memory. When a page is downloaded, it is written to the flash memory. To adapt to the limited RAM space on sensor nodes, pages are updated in strictly increasing order. At any time, different nodes may be downloading different pages while the nodes that are closer to the sink node are more likely to be updating a large number of indexed pages.

As shown in Figure 39, Deluge uses a simple **advertise-request-data** handshaking strategy to support fast code update. At the beginning, the states of all sensor nodes are set to the **Maintain** state. Each sensor node periodically broadcasts the advertisement messages (**ADV**), which contain the information of the application code that it has. When a sensor node **S** receives an advertisement that indicates that the neighbor **N** has a newer version of the application, or has the same version but newly downloaded pages, node **S** will send out a request message (**REQ**) to **N** to request a new page, and change its own state from **Maintain** to **Request**. The state will be changed back when it receives all the packets in the requested page. Node **N** changes its state to **Transmit** when it receives the request message from **S**, and starts sending all the packets of the requested page to node **S**. After sending the page, **N** changes its state back to the **Maintain** state.

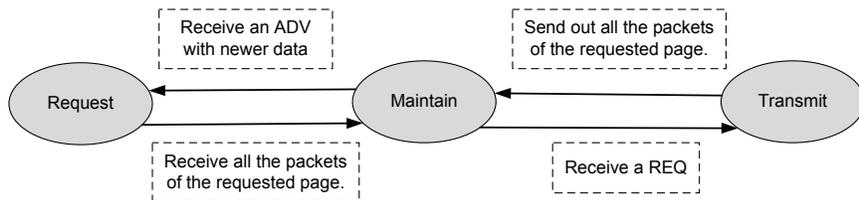


Figure 39: Advertise-request-data handshaking protocol in Deluge.

To ensure the received code is sent from the trusted sink node and has not been modified by malicious attackers, the packets may be encrypted and/or authenticated for security protection [23, 41, 62].

5.1.2 Melete: a controlled broadcasting protocol for MA-WSNs

Software switch differs from software upgrade in that: (i) only a subset of the sensors need to switch to run a particular application, and some of them may have the code; (ii) in addition to the sink node, other sensors that have the requested application can also be the source nodes for code distribution. Figure 40 shows a software switch example. Three applications are distributed across different nodes in a network. The code distribution problem arises when there is a need to reprogram some nodes to run application A.

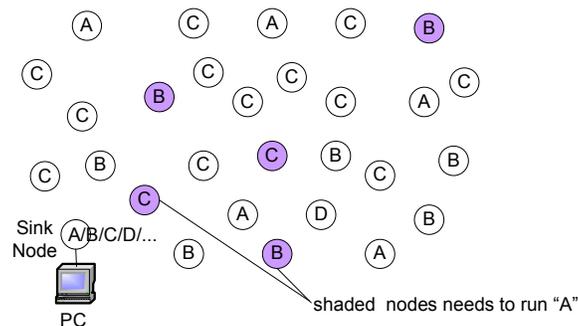


Figure 40: An example of software switch in a multi-application WSN (MA-WSN).

There are two existing approaches. A naive solution is to directly apply Deluge and disseminate application A from the sink to all sensors. After dissemination, the nodes that do not need A discard the code from their storage. The solution is clearly not a good choice due to unnecessary packet transmissions to the nodes that do not need it. The other solution is to let requesting nodes initiate code dissemination and fetch A from nearby sensors. Melete [83] is such a protocol — the nodes that need to run A broadcast their requests within a controlled range and discover the source nodes that have A. Sources then send back the requested data packets. However, as a stateless protocol, Melete does not record the source nodes and has to discover them repeatedly. When transmitting applications with multiple pages, multiple sources within the range may respond and thus cause congestion due to signal collision.

5.2 MCP: A MULTICAST-BASED CODE REDISTRIBUTION PROTOCOL

5.2.1 An overview of the protocol

In this section, I propose a multicast-based code redistribution protocol, MCP, to solve the “n to n” code distribution problem in software switch. MCP employs a gossip-based source node discovery strategy. Each sensor summarizes the application information from overheard advertisement messages, and stores this information in a local application information table (AIT). Future dissemination requests are forwarded to nearby source nodes rather than flooding the network. Different from the Deluge [32] and Melete [83] schemes discussed above, the data messages are only multicast to the requesters, which avoids unnecessary packet transmission in the network. Using AIT, the request messages can be directly sent to the source nodes, avoiding request message flooding in the network.

An overview of this protocol is as follows.

- Sensors in MCP periodically broadcast ADV messages to advertise their knowledge about running applications in the network, which is similar to Deluge. Each sensor summarizes in an *application information table (AIT)* the ADV messages that it overhears from the network.
- To reprogram a subset of sensors, the sink floods a dissemination command that guides which sensors should switch to run application A. For example, a command “[B→A, p=0.25]” from the sink node indicates that the sensors whose active application is B should switch to A with a 25% probability. That is, 25% of the nodes that are currently running application B will switch to A.
- After receiving the command from the sink, each sensor identifies its dissemination role as one of the following.
 - (i) a *source* if the sensor has the binary of application A;
 - (ii) a *requester* if the sensor does not have the binary of A but needs to switch to run A;or
 - (iii) a *forwarder* if the sensor is neither a *source* nor a *requester*.
- To fetch all pages of application A, each *requester* periodically sends out requests (i.e.,

REQ messages) to its closest source. While the AIT table records up to three nearby sources, only the closest one is tried. The REQ messages are sent to the source via multicast without flooding all nodes within the range. If no response was received before timeout, a requester tries to resend the REQ message. The requester tries each source node several times before marking it as a *temporary non-available* source. More details will be elaborated in the following section.

- A source node responds with the data (i.e., Data messages) that contain code fragments while a forwarder forwards both request and data packets.

Similar to Melete and Deluge, MCP uses three types of messages: an ADV message that advertises interesting applications; a REQ message that asks for packets of a particular page; and a Data message that contains one data packet (i.e, a piece of code segment).

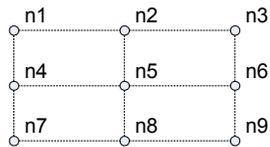
5.2.2 ADV message and application information table (AIT)

In MCP, each sensor periodically broadcasts ADV messages, and summarizes the information of overheard ADV messages into a small application information table (AIT). The AIT table serves not only as an application-dependent routing table, but also as a small database to track the application versions in the network. An example is shown in Figure 41.

Each ADV message contains the information of one application: (i) an application ID and version number; (ii) the number of pages of the application; (iii) the information of the two closest source sensors — the source ID and number of hops to the source (S, H); and (iv) the CRC checksum. If a sensor has multiple known applications, it advertises them in a round-robin fashion to reduce the broadcasting overhead. Note that a sensor may not have space to store the code images of all its known applications.

The AIT table summarizes the overheard ADV messages. In addition to the application summary, AIT stores up to three closest source nodes for each known application, and the uplink sensor ID for each source, i.e., from which the source information was received. I keep three source nodes because keeping more is not necessary. As wireless communication is expensive, a requester prefers to use its closest source and try others only when the closest one is unavailable (e.g., busy). Using two backup sources provides enough tolerance such

Network: Assume n1 has A1; n3, n5 n9 will change to A1



On Node N9:

Application ID	version	# pages	node ID	hop #	uplink ID
A1	1	8	n1	4	n8
			n3	2	n6
			n5	2	n8
A2	1	8
		
		

On Node N4:

Application ID	version	# pages	node ID	hop #	uplink ID
A1	1	8	n1	1	n1
			-	-	-
			-	-	-

Figure 41: An example of the application information table (AIT).

that the requester does not need to wait to discover a new source if one or two known sources become unavailable.

Each AIT entry occupies 12 bytes — the application ID has 4 bits; the version number has 4 bits; the page count has 10 bits; each node ID takes 10 bits (to support a thousand-node WSN); and the hop count has 6 bits (to identify source nodes within 64 hops). Assuming there are 10 applications in a network, the AIT table is only 120 bytes. I place the table in the data memory for fast access.

When an incoming ADV message contains new information, the corresponding entry in the AIT is updated. Assume a sensor S_1 receives an ADV message from S_2 , and the message identifies two nearby sources (S_3, H_3) and (S_4, H_4) where H_3 and H_4 indicate the number of hops from S_2 to sources S_3 and S_4 . If S_1 already records the information of three sources (S_5, H_5, U_5) , (S_6, H_6, U_6) , and (S_7, H_7, U_7) , then it updates the AIT table according to the following rules.

- If one entry in the AIT table records the previous message from the same uplink S_2 and it refers to the same source, e.g., $S_5=S_3$ and $U_5=S_2$, then the information in the ADV message represents the up-to-date source information and replaces the old entry.
- If one entry in the AIT records a longer path to an advertised source, e.g., $S_5=S_3$, $U_5 \neq S_2$, and $H_5 > (H_3+1)$, then the hop count and uplink node from the ADV message replace those in the AIT.
- If the advertised source cannot be found in the AIT, and there is an invalid entry in the table, then the new source is inserted into the table.
- If the ADV message advertises a closer source than one of those in the AIT table, then the closer source replaces the farthest source in the AIT.

A sensor advertises the applications in the AIT in a round-robin fashion, and prioritizes the applications whose entries have been recently updated: (i) the applications whose sources were recently updated are advertised before those that were not; (ii) in one round, the applications whose sources were recently updated are advertised three times while others are advertised once. In addition to normal ADV advertisement, an application is advertised if the sensor receives a broadcast request for that application, as I elaborate next. If there are

several updated applications, MCP updates them sequentially as I discussed in Chapter 1. Due to sequential update, once a sensor detects an application being updated, it halts the advertisement of other applications and advertises only the one being updated.

5.2.3 Request multicasting

In MCP, a requester periodically sends out request messages until it receives all pages of the target application. Given the target application, the requester searches the AIT for the closest alive source and constructs a REQ message as follows.

$$\text{REQ} = [\mathbf{S}, \text{pgNum}, \text{bv}, \text{Rv}]$$

where \mathbf{S} indicates the selected source node, pgNum indicates the current working page; bv is a bit vector indicating the requested packets in the page; and Rv is a routing vector used to support GRADient broadcast routing [82]. If the AIT records more than one source node, the requester always tries the closest source first and forms a gradient routing region as shown in Figure 42.

MCP follows the GRADient broadcast routing design in [82]. The routing vector Rv contains four fields: (i) field α is the amount of credit assigned for routing; (ii) field \mathbf{C}_s is the hop count to the selected source \mathbf{S} — it is the value recorded in the AIT table; (iii) field \mathbf{P}_c is the number of hops when reaching a forwarder node; (iv) field \mathbf{C}_r is the the hop count from the forwarder node to \mathbf{S} . When a packet reaches a forwarder node, the consumed credit is calculated as $(\mathbf{P}_c + \mathbf{C}_r - \mathbf{C}_s)$. In our experiments, we choose α to be 2 and it works well. More details can be found in [82].

A requester continues sending REQ messages when it cannot finish the page before time-out. After three tries, it marks the source that it tried to reach as an *unreachable* source. If the AIT does not record any nearby alive source, then the source sets to be *null*, indicating the REQ message is flooding all nodes within k hops. k is initially set to 4; if k -hop flooding cannot find a source, MCP doubles k and tries again. After receiving a broadcast request, an idle forwarder within the range forwards the request; an idle source node within the range always responds with requested packets. A node is defined as *idle* if it is not involved in serving any requester (either as a forwarder or a source).

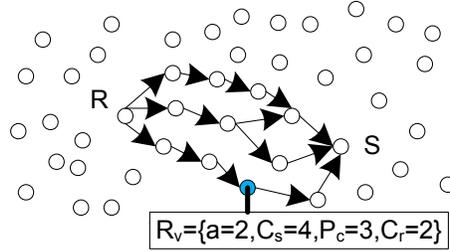


Figure 42: Gradient-based request routing. (R:requester, S: source).

Since each requester sends out REQ messages independently, different requesters may work on different pages. MCP allows node preemption. If a REQ message asking for page x reaches a working node W that is currently working on page y , and $x+1 < y$, then the node W quits the current state and switches to serve the request. If W is a forwarder, then it forwards the request; if W is a requester or a source, then it must have the requested page and thus will respond with the requested packets. The node enters the idle state after serving the request. By allowing preemption, multiple requesters are working on the same or close-by page numbers, which creates opportunities for caching, as we discuss next.

5.2.4 Caching

During code dissemination, some requesters or forwarders, while working on the current page, may overhear packets from pages with larger indices. As code pages are requested strictly in increasing order, a requester will ask for these pages, and a forwarder has a high possibility to receive requests for these pages.

To improve energy efficiency, sensors in MCP buffer such packets in their data memory. The space that can be dedicated to caching on a wireless sensor is usually very limited. While it is possible to exploit external flash for caching, accessing external flash has many drawbacks. I did not use flash for caching in this dissertation.

- Flash writing speed is slow. It takes about $78\mu\text{s}$ to finish writing one byte to the flash. As a comparison, it takes about $32\mu\text{s}$ to transmit one byte on MICAz nodes [63]. Flash

caching may miss receiving some packets and increases retransmissions;

- Flash writes consume significant energy. It requires $3\mu\text{J}$ and $1.5\mu\text{J}$ to write one byte to the flash and transmit one byte respectively [63];
- Flash writes have to be done at the block level, e.g., 256 bytes on MICA2 nodes. To cache the content of one received packet (23 bytes), the sensor has to read the corresponding block, modify it in memory, and then write it back. Clearly this is very energy-inefficient.
- Flash memory usually can sustain about 10,000 writes during its lifetime. Too many writes may shorten the network lifetime.

Caching on a requester is straightforward — the sensor caches the next several pages in addition to the current working page. However, it is slightly more complicated on a forwarder node as it gets requests from different requesters that work on different pages and may suffer from thrashing if it takes turns to serve these requests. In MCP, a forwarder gives priority to pages with smaller indices. The cached pages are periodically cleared.

6.0 EXPERIMENTAL RESULTS

This chapter evaluates my proposed software update management framework. I will first present the experiment methodology, and in particular, how to compose a set of update benchmarks for evaluation in the dissertation. I will then present and analyze the experimental results in two groups — (i) For the results collected from code dissemination in SA-WSNs, since the same dissemination tool (Deluge) is used for different schemes, I focus the analysis on the generated script size, the code performance, and the overall energy efficiency; and (ii) For the results collected from code dissemination in MA-WSNs, since the challenge comes from the dissemination at the network level, I focus the comparison on the network traffic and time consumption between the proposed multicast-based code distribution protocol and other existing protocols.

6.1 CONSTRUCTION OF THE UPDATE BENCHMARK SUITE

Software update management is a new topic in the wireless embedded system community. While some manually constructed in-house update samples were used in published works [22, 35, 68], no benchmark suite is publicly available. To thoroughly evaluate the effectiveness of my proposed framework, I took the effort to construct a sensor software update benchmark suite that covers the software update cases for both general-purpose applications and DSP applications in WSNs.

My software update benchmark is constructed on a set of base programs, listed in Figure 43. These base programs come from three sources: (i) the sample sensing applications included in TinyOS [76], an open-source OS designed for WSNs; (ii) AES encryption code

included in CryptoLib, a security algorithm library; and (iii) the DSP applications included in the DSPstone [21] benchmark suite.

Base 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 the AES algorithm. I select the encryption code in the experiment.
Deluge	TinyOS	The mulithop code dissemination protocol in TinyOS. I tracked its continuous updates in different TinyOS versions as a real-life case study.
Matrix	DSPStone	Matrix multiplication.
ADPCM	DSPStoe	A waveform codecs using adaptive differential pulse code modulation.

Figure 43: Base programs for the construction of the software update benchmark.

6.1.1 Test case categorization

Categorization based on update levels. Each test case consists of two programs — an old program A and a new program B. Based on the portion of the difference between A and B, the test cases can be categorized as

- (a) Low-level updates represent those that change only one or two basic blocks;
- (b) Medium-level updates represent those that change multiple places within a large function or across several functions, but still preserve the overall structure of the original code;
- (c) High-level updates represent those that significantly change the code structure. This often means changing from one application to a different application.

Frequent updates such as code fixes and sensor reconfigurations are mainly low to medium-level changes, while application functionality change falls in medium or high-level changes.

Categorization based on how they are constructed. Based on how the test cases are constructed, I categorize them in three categories and discuss them in more detail in the next section.

- (a) Real test cases. They are the ones directly extracted from the base benchmark programs.
- (b) Manually constructed test cases. They are the ones that I constructed to test specific properties of the framework.
- (c) Automatically constructed test cases. They are the ones that are constructed automatically for strength test.

6.1.2 real-bench: real test cases

In our base benchmark, some applications have multiple versions that naturally form a set of real-world case studies. Within the 15 releases of TinyOS-1.x, the default code distribution protocol Deluge undergoes a couple of updates. The updates range from one-line bug fixes, to feature enhancements, to completely reconstructing the code. I collected a mix of these real code update cases. In addition, some DSP applications implement the same functionality using alternative algorithms that approximate another type of software update scenarios.

I did not include the large update where TinyOS-1.x changed the code distribution protocol from MNP to Deluge. For such a large update, I did not find much benefit in using the algorithms proposed in my framework. Luckily, the framework can identify whether there is any performance gain beforehand and roll back to the traditional workflow if there is no benefit to achieve, i.e., compile to get the new image using `gcc`, and distribute the complete image.

General-purpose application update benchmark. Figure 44 illustrates the details of the real test cases when Deluge [32] undergoes a series of updates from TinyOS version 1.52 to version 1.58.

DSP application update benchmark. For the DSP applications, I selected the matrix multiplication function `matrix.c` and one function in the ADPCM implement `speed_control` as the real DSP update benchmarks. More information can be found in Figure 45.

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

Figure 44: Real general-purpose application update benchmark.

Case#	Function & Versions	Update Level	Description
R-D-1	matrix1.c \Rightarrow matrix2.c	Medium	Move two iterations out of the loop.
R-D-2	speed_control 1 \Rightarrow 2	Medium	Seven temporary variables are introduced to hold the value of the comparison results.
R-D-3	speed_control 2 \Rightarrow 3	High	Multiple global variables are combined into a structure. The references to the variables are changed due to this change.

Figure 45: Real DSP application update benchmark.

6.1.3 man-bench: manually generated test cases

While real test cases show real-world update patterns in the system, only a small number of them can be collected. This is not enough to thoroughly evaluate my proposed framework. To expose the properties of my proposed algorithms, I manually constructed a mix of updates ranging from small to large update levels. When I constructed these test cases, I tried to keep the update meaningful. For example, an inserted variable is always used; and an inserted `if`-statement does use some variables to make a decision and include some statements in its branch paths.

Since an important component of my dissertation is update-conscious compilation, I constructed test cases to cover possible updates related to program structures, including variable insertion/deletion, instruction insertion/deletion/update inside and outside loops, and the control flow insertion/deletion/update.

General-purpose application update benchmark. The TinyOS applications shown in Figure 43 are selected as the base benchmarks to create the manually generated general-purpose software update benchmark. Figure 46 summarizes the synthetic updates made to these benchmarks. The updates vary from low-level to high-level changes.

DSP application update benchmark. For the DSP applications, I inserted/deleted code to create/remove the variable interferences to the DSP base benchmarks, such as the matrix multiplication function *matrix.c* and one function in the ADPCM standard implementation *speed_control* as the real DSP update benchmarks. The detailed benchmarks are listed in Figure 47.

6.1.4 auto-bench: automatically generated test cases

The test cases in the previous two groups have limitations. The number of real test cases is small, while the manually generated test cases may be biased due to personal experiences. In this section, I wrote a tool to automatically construct another mix of test cases.

General-purpose application update benchmark. For general-purpose application study, the generated test cases are used to evaluate the compilation time of the update-conscious compilation schemes. The compilation time here depends on the complexity of

Case #	Function	Update Level	Update details
M-G-1	CntToLeds	Low	Change the color of blink.
M-G-2	Blink	Low	Insert one local variable and one use in run_next_task.
M-G-3	AES	Low	Insert one local variable and use it within the loop in aes_encrypt.
M-G-4	AES	Low	Change one instruction in aes_encrypt.
M-G-5	AES	Low	Insert a local variable in aes_encrypt and use it twice — within and outside the loop.
M-G-6	Blink	Low	Add a new parameter in TOSH_run_task.
M-G-7	CntToLeds	Medium	Insert three variables and their uses;
M-G-8	CntToRfm	Medium	Insert a global variable and use in three different functions.
M-G-9	CntToRfm	Medium	Insert a local variable and use it several times in TOSH_run_next_task function.
M-G-10	Blink	Medium	Insert a global variable and use it in a new if/then branch in TOSH_run_next_task function.
M-G-11	Blink	Medium	Add an else branch for an if statement in Timer_HandleFire.
M-G-12	CntToRfms ⇒ CntToLedsRfm	high	Change the application from CntToRfms to CntToLedsRfm
M-G-13*	CntToLeds⇒ CntToRfms	high	Change the application from CntToLeds to CntToRfms.
M-G-14	AES	Medium	Add two and remove one local variable in function invShiftRows().
M-G-15	AES	Medium	Add one and remove two local variables in function invShiftRows().
M-G-16	AES	Medium	Add one local variable in function invShiftRows() and add a four-element array in function invMixSubColumns().
M-G-17	AES	Medium	Remove one local variable in function invShiftRows() and remove a four-element array in function invMixSubColumns().
M-G-18	AES	High	Remove one and add two local variables in function invShiftRows(). Remove two and add four local variables in function invMixSubColumns().
M-G-19	AES	High	Add one and remove two local variables in function invShiftRows(). Add two and remove four local variables in function invMixSubColumns().

*: The experimental results of this case are shown in the text instead of the graphs to make the graphs more proportionally precise.

Figure 46: Manually generated general-purpose application update benchmark.

Test Case	Function	Update Level	Description
M-D-1	verify.c	Low	Update one basic block to create the interference between two variables that are not coalesced in the original version.
M-D-2	verify.c	Medium	Update one basic block to create the interference between three variables that are coalesced in the original version.
M-D-3	verify.c	Medium	Expand the live ranges of three variables to cross basic blocks .
M-D-4	matrix1.c	Medium	Shrink the live range of the one variable and Expand the live range of another variable within one basic block. Over ten interferences are updated.
M-D-5	matrix1.c	Medium	Shrink the live ranges of the two variables and Expand the live ranges of another two variables within one basic block. Over ten interferences are updated.

Figure 47: Manually generated DSP application update benchmark.

the ILP problem created by the update-conscious compiler. Because the number of decision variables, constraints, and the complexity of the objective goal all affect the problem complexity, one source-level modification may create problems with quite different complexity levels depending on the type of the modification and the place where it is made. Thus, instead of modifying the source code, I created the benchmarks by modifying the intermediate representations directly. Random intermediate representation statements are inserted to or removed from the intermediate representation of the base benchmark. Given the number of intermediate representation statements to be modified, I created multiple cases to show the bound of how this affects the problem complexity.

DSP application update benchmark For DSP application study, the automatically generated test cases are used to evaluate the compilation performance, including the patch script size deduction and the run-time execution overhead. Because the direct factors are the memory access sequence and the interference between each pair of variables, I created the automatically generated benchmarks by directly modifying these two factors.

6.1.4.1 Methodology used to generate the auto-bench

Although different tools are needed to automatically construct the test cases for general-purpose applications and DSP applications respectively, the methodology is the same.

First, the tool always takes the intermediate representations (IRs) as input. For general-purpose applications, the IRs of a base application are used directly. For DSP applications, the access sequences and the variable interference are used. Since only these two affect the algorithms in my dissertation, they are at the same level as IRs. Second, the update percentage is defined as the number of changed IRs to the total count. Given a percentage to update, the tool randomly determines that many updates, while each update randomly chooses a place and randomly decides whether to insert, remove, or update the corresponding IR. The tool generates 500 instances for each update percentage, and the averaged results are reported in this chapter.

While `auto-bench` helps to perform the strength test, it has limitations. Currently, the tool is preliminary — some generated test cases may be semantically inappropriate. For example, a variable may be used but not initialized. The tool has been upgraded several times, and many straightforward incorrect cases have been removed, e.g., the one that defines a variable but never uses it. To summarize, this group of test cases are designed to identify the trend by studying a large number of instances.

6.2 PATCH GENERATION AND DISSEMINATION IN SA-WSNS

I have implemented the proposed update-conscious register allocation and data allocation schemes. In this section, I will present my experimental settings and discuss the results on code quality, energy efficiency, and compilation time.

6.2.1 Updating general-purpose applications using UCC-RA

In order to compare the performance of the proposed update-conscious compilation register allocation scheme (UCC-RA) with GCC-RA, I used the manually generated general-purpose

benchmarks (M-G-1 ~ M-G-13) listed in Figure 46 to generate the binary images and further the patch scripts. Then I used automatically generated general-purpose benchmarks to study the problem complexity and compilation time.

6.2.1.1 Settings

I simulated a sensor network that consists of Mica2 mote nodes [19] running TinyOS [76]. The processor that Mica2 (MPR400CB model) uses is the AMTEL AVR micro controller — ATmega128L [3].

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

I selected `Avrora`, an instruction-level sensor network simulator, to collect the execution cycles of the code before and after compiling the updated code with a UCC and GCC (the accuracy of the simulator has been reported in prior work [77]). I then plugged in the energy model and execution profiles to study the energy consumption tradeoffs with different compilation approaches.

6.2.1.2 The generated script size

Figure 46 summarizes the synthetic updates that I made to the benchmarks. The updates vary from low-level, through medium-level, to high-level changes, as described below:

- The low and medium-level changes cover a wide range including constant changes, variable changes, parameter changes, instruction changes, 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, I only reported the changes in the functions that

are directly affected (rather than, for instance, code shifting due to expansion/shrinkage of neighbor functions). In addition, I observed minimal inter-procedural correlation. For example, the same global variables are assigned to different registers in different functions. Therefore, the overall impact of high-level updates is close to the summarized changes of simple updates.

- I evaluated 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.

I first conducted experiments to compare the generated script size between UCC-RA and GCC-RA. For GCC-RA, I manually find the best match between the new and old binaries. This is the lower bound of existing `diff`-based code dissemination algorithms [60, 68]. That is, I compared my results against the best possible implementation of existing update-unconscious approaches [60, 68].

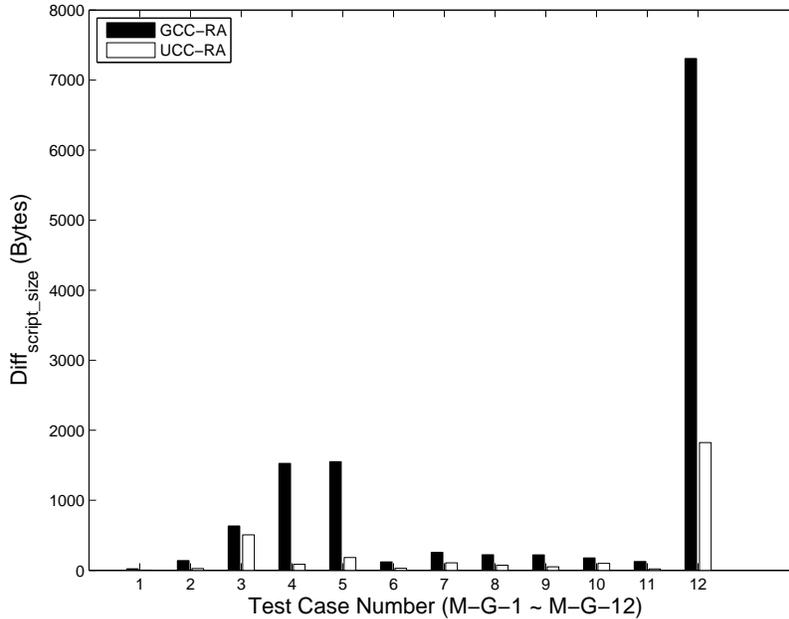


Figure 48: Script size comparison between UCC-RA and GCC-RA.

Figure 48 shows the results, in $Diff_{script_size}$, for update test cases M-G-1 ~ M-G-12. From the figure, 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 M-G-1), but may also propagate local changes to

a much larger range (test case M-G-4).

I then studied the two high-level changes. M-G-12 introduces several new functions most of which are small *inline* functions. They disturb the register selection in a large function and introduce a significant number of differences, which are seen when using GCC-RA. Fortunately, those differences are minimized in UCC-RA. Test case M-G-13 represents another type of high-level changes, and the application `CntToLeds` is quite different from `CntToRfms`. The former has 828 instructions while the latter has 4351 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 3929 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`).

Therefore, it is beneficial to apply update-conscious compilation for small to medium-level changes. The benefits diminish when the update is large.

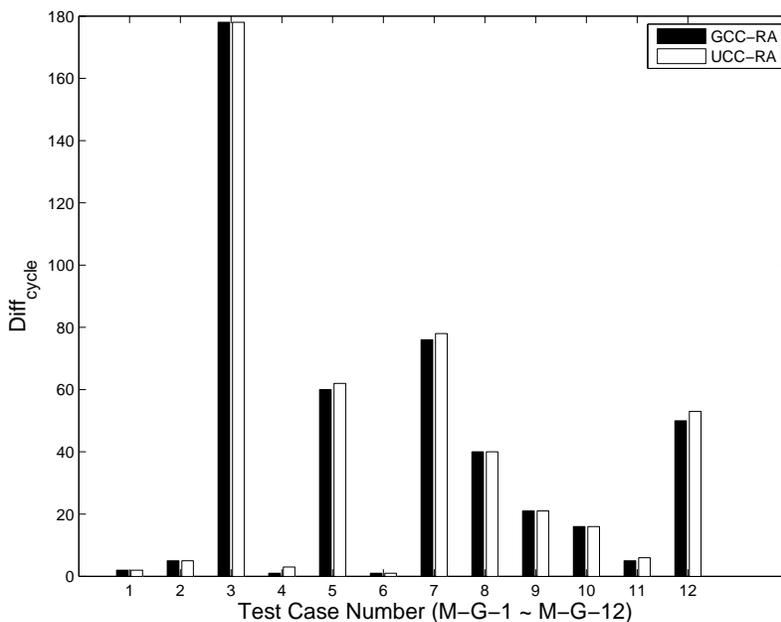


Figure 49: Code quality comparison between UCC-RA and GCC-RA.

6.2.1.3 The generated code quality

I 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 binary. This metric also indicates the slowdown after applying update-conscious compilation.

Figure 49 shows the results for test case M-G-1 \sim M-G-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 code needs to be generated. Thus, register conflicts are small. In some cases, e.g., test case M-G-12, UCC-RA inserts three `mov` instructions since by doing so, it can save 406 instruction updates.

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

For test case M-G-13 (not shown in the figure), UCC-RA uses the preferred register tag as a hint when selecting registers. It reaches the same allocation result and thus has the same code quality as the one generated by GCC-RA.

6.2.1.4 The energy savings

The energy savings are calculated as follows. I first compute $Diff_{energy}$ (defined below), 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, I compute the energy savings per update for GCC-RA and UCC-RA respectively.

$$Diff_{energy} = Diff_{script_size} \times E_{trans} + Diff_{cycle} \times E_{exe} \times Cnt \quad (6.1)$$

$$EnergySavings = Diff_{energy}^{GCC-RA} - Diff_{energy}^{UCC-RA} \quad (6.2)$$

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.

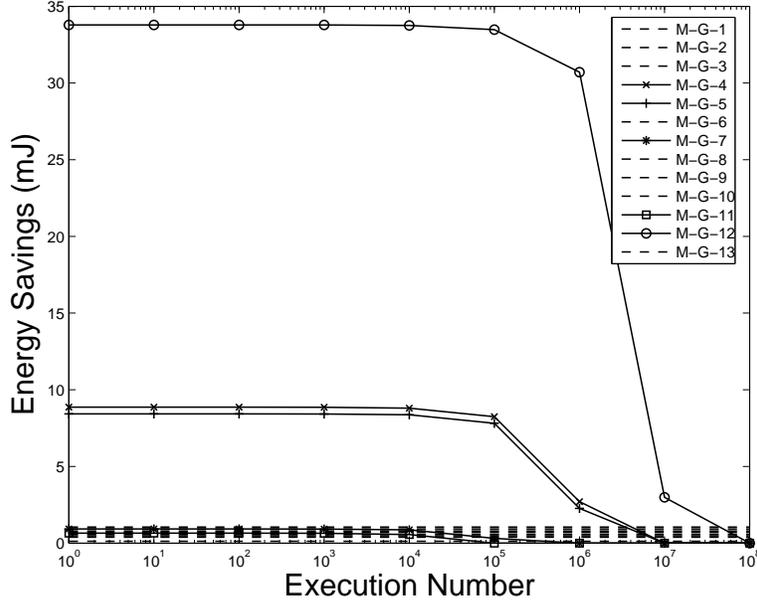


Figure 50: The energy savings for general-purpose applications.

Figure 50 plots 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, 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 the number of instructions I reduce from GCC-RA, the less data I need to transmit, and the more savings I gain from UCC-RA.

When the code generated from UCC-RA runs slightly slower than that from GCC-RA (e.g., test case M-G-12), extra energy will be consumed in instruction 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, the breakeven point for test case M-G-12 is 10^7 times. Given that a Mica2 sensor can last for about one year if it stays active for about 15 minutes per day, the code

generated from UCC-RA saves energy unless the code is executed about 30 times per second ($=10^7 \div 365 \div 15 \div 60$) for its whole lifetime. This is impossible because the target application `CntToLedsRfm` of M-G-12 converts counters to light blinks, and human eyes cannot tell if a light blinks 30 times a second (not to say the device cannot blink at this high frequency).

6.2.1.5 The problem complexity and compilation time

A concern of ILP is its slow compilation time. I measured the compilation time for each test case and the result is shown in Figure 51. The larger the update is, the longer it takes to re-compile it. For instance, M-G-12 is the largest update in the test set and it takes over 20 seconds to compile. Of course, when the update is larger, more decision variables and constraints will be generated by UCC-RA, which makes the ILP problem harder to solve, so the whole compilation takes longer to finish.

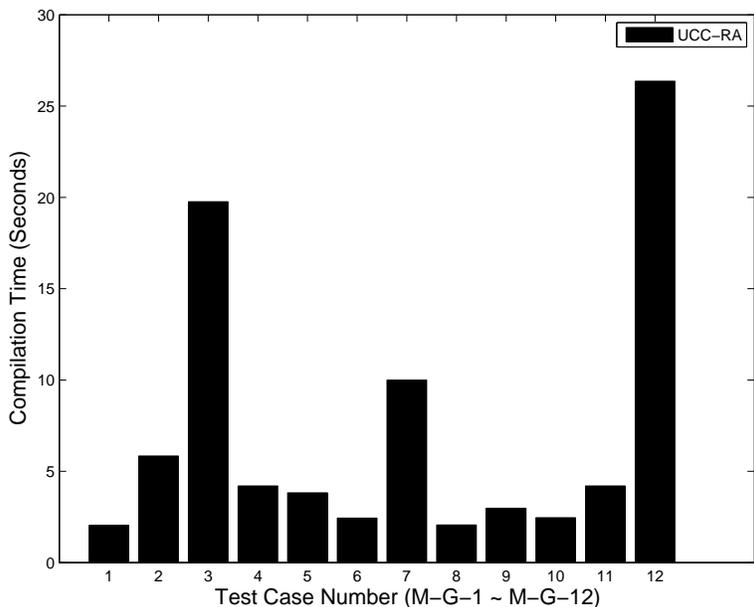


Figure 51: The compilation time of UCC-RA.

To study the compilation time of general cases, I used `auto-bench` to evaluate the ILP problem complexity as a function of the number of constraints and the number of decision variables.

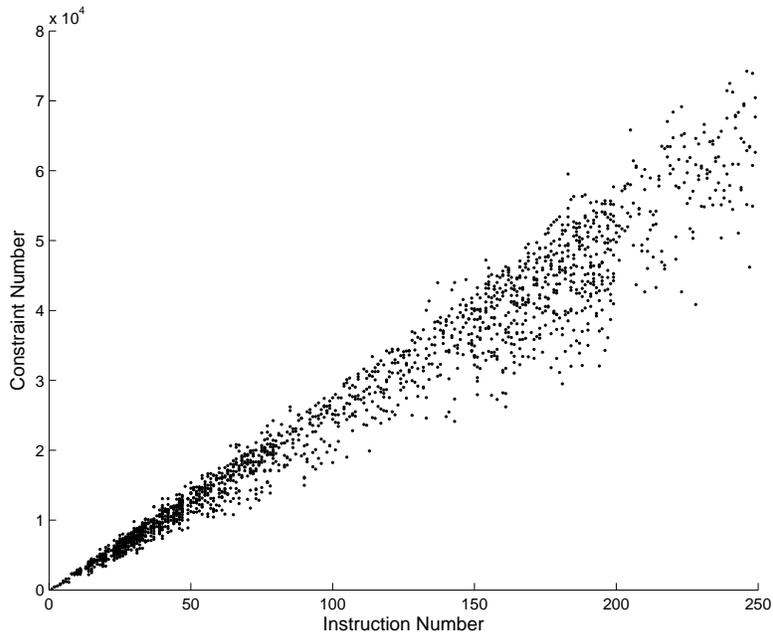


Figure 52: The number of constraints as a function of the number of IR instructions.

Since the ILP problem is more complex to solve when the number of instructions (from a changed code segment and needed to recompile) increases, Figure 52 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. I plot the number of iterations that the LP_solve [8] requires as a function of (the number of variables \times the number of IR instructions) in Figure 53.

An interesting observation in my experiments is that the preferred register tag helps to improve the performance. Compared to an ILP-based register allocator which allocates registers from scratch, the preferred register tag is a hint to the solver and can reduce the number of iterations that the solver needs to try. As an extreme case, I also tested misleading preferred register tags, e.g., variables are assigned to the preferred register tag randomly, and found that the solver may need 2 or 3 times more iterations to solve.

To see how fast the problem can be solved, I conducted timing experiments on an Intel Xeon 3.6GHz processor running Fedora Linux 2.4.21 kernel. The physical memory size is 2GB

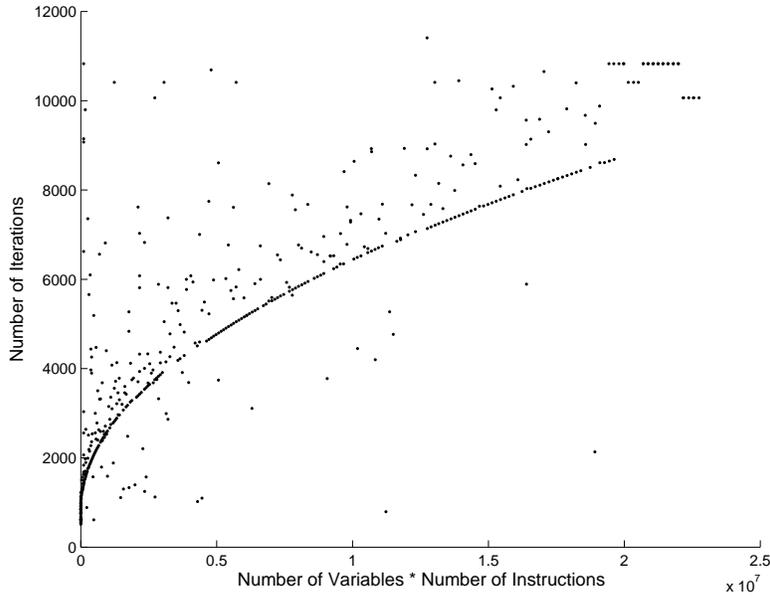


Figure 53: The number of iterations as a function of (the number of variables \times the number of IR instructions).

while in the experiments, the largest observed memory usage is less than 256 MB. Figure 54 shows that the average time required to solve one iteration increases about linearly with the problem complexity. It usually takes the solver less than 3 minutes to allocate registers for a chunk of 250 IR instructions. As a comparison, it takes GCC-RA far less than one 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 (in Chapter 3.2.2 I discussed the reason why only changed chunks are considered in my framework); (iii) it is worthwhile to trade the compilation time at the server side (the sink node) for the energy savings on sensor nodes. As discussed in Chapter 1, I assume the sink has no or negligible resource constraints while sensor nodes have very tight resource constraints.

I also examined whether approximating the original non-linear integer programming problem with a linear problem degraded the final results. I observed the same allocation decisions for all the test cases with or without the approximation. The only difference is

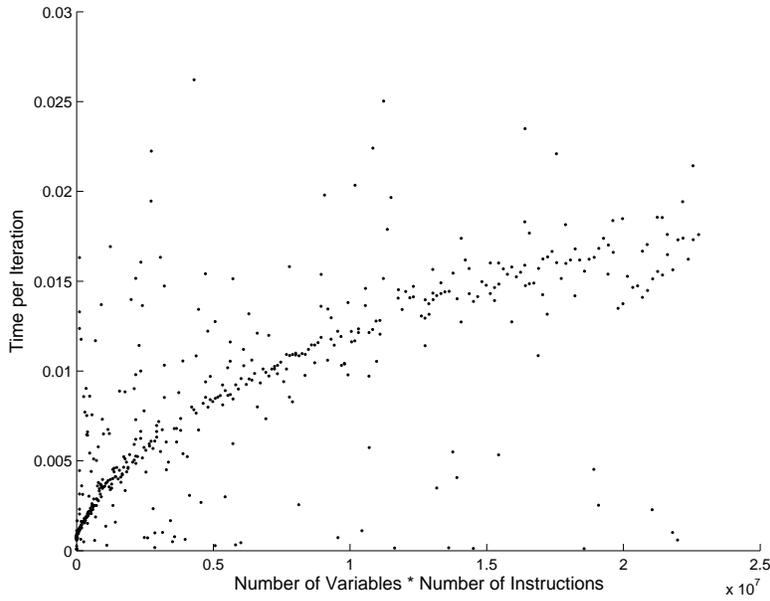


Figure 54: The time to solve one iteration as a function of (the number of variables \times the number of IR instructions).

that solving non-linear problems is orders of magnitude slower than a linear problem.

6.2.2 Updating general-purpose software using UCC-DA

In order to compare the performance of the proposed update-conscious compilation data allocation scheme (UCC-DA) with GCC-DA, I used the manually generated general-purpose benchmarks (M-G-14 \sim M-G-19) listed in Figure 46 to generate the binary images and further the patch scripts.

The tradeoff of UCC-DA is between the stack size and the generated script size. Keeping variables in the same locations as those in the old version reduces the update script size. However, the stack size grows faster, as some slots are wasted. In this section, I evaluated the reduction of script size under a tight threshold `SpaceT`, i.e., the number of memory slots that can be wasted at runtime.

In the experiments I set `SpaceT` to be 5. The reason that I chose a relatively small threshold is that the test cases are relatively small, and choosing a larger constant would

diminish the need for variable relocation and give too much freedom to UCC-DA. The biased conclusion assuming no need for variable relocation may not be applicable to real applications.

6.2.2.1 Settings

To get the baseline binary I used `ncc`, the NesC compiler included in TinyOS release, and `avr-gcc`, the GNU C compiler (GCC) re-targeted for AMTEL AVR micro controllers. The generated binary is compared with the older version to generate the update script. I compared the generated script size between GCC-DA and UCC-DA.

Then, I used `tos-ramsize` [67], the tool included in the TinyOS release, to generate the worst-case stack size of the binary generated using different data allocation schemes.

The update benchmarks are listed as M-G-14 ~ M-G-19 (Figure 46). They are constructed from the Advanced Encryption Standard (AES) application [69]. It takes several steps to encrypt or decrypt the given data, and in each step it does some relatively complicated computation to get a temporary result, and feeds it to the next step. For example, in the `ShiftRow` step, it cyclically shifts the bytes in each row by a certain offset. Local variables are heavily involved here to store the temporary results.

6.2.2.2 The generated script size

I first used UCC-DA and GCC-DA to generate the new binary, and compared it with the old one to generate the update script. Figure 55 shows the script size comparison.

Using UCC-DA, the script size can be reduced by 56% on average. New variables are always allocated at the top of the stack no matter where they are declared, so that the unchanged variables that are declared after these new variables do not have to be relocated. For the deleted variables, the memory holes are left unfilled, if the total wasted memory size is smaller than the threshold `SpaceT`. Otherwise, the variable are selected to fill up the holes. Under the given threshold `SpaceT`, the UCC-DA algorithm keeps most of the unchanged variables in their original memory location, so that it minimizes the update to the memory access instructions that access those variables.

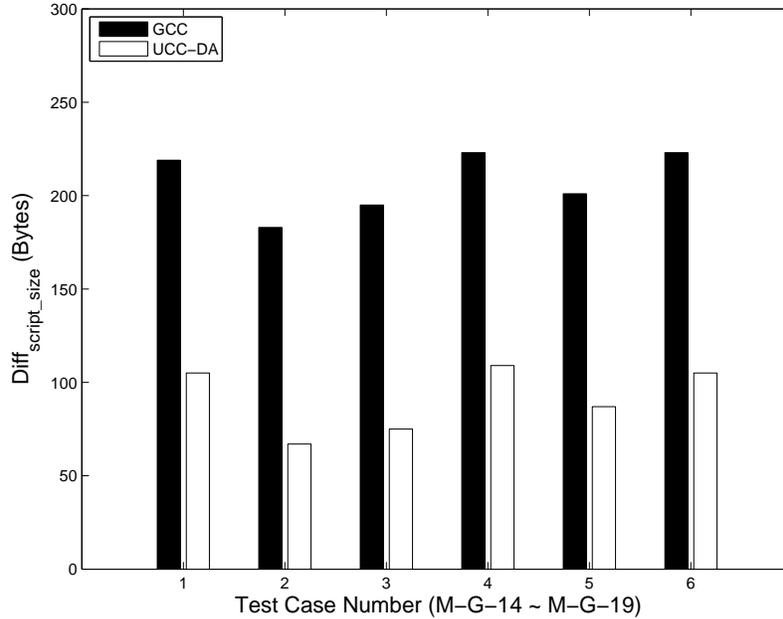


Figure 55: Script size comparison between UCC-DA and GCC-DA.

6.2.2.3 The energy savings

Figure 56 shows the energy savings using UCC-DA. As we can see from the figure, the results are independent of the execution frequency. This is because no extra instructions were introduced in the binary. The energy savings come from sending a smaller update script to the remote sensors.

6.2.2.4 The wasted memory space

The UCC-DA algorithm trades RAM usage for script size reduction. Keeping the unchanged variables in place may cause memory holes if some variables are removed. Thus, it may waste some memory space at runtime, which results in a larger worst-case stack size. Figure 57 compares the worst-case stack size of the binaries generated by different algorithms.

From the experiment results, using UCC-DA only increases the memory usage by 1.9% on average, yet it reduces the script size by 56%. This is because the memory space may be wasted only when the memory space taken by the removed variables is larger than the

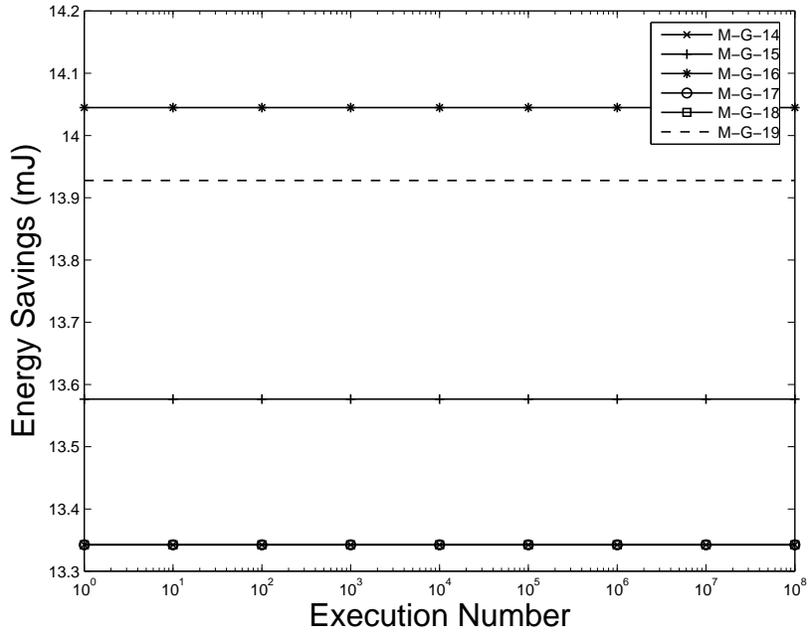


Figure 56: Energy savings using UCC-DA.

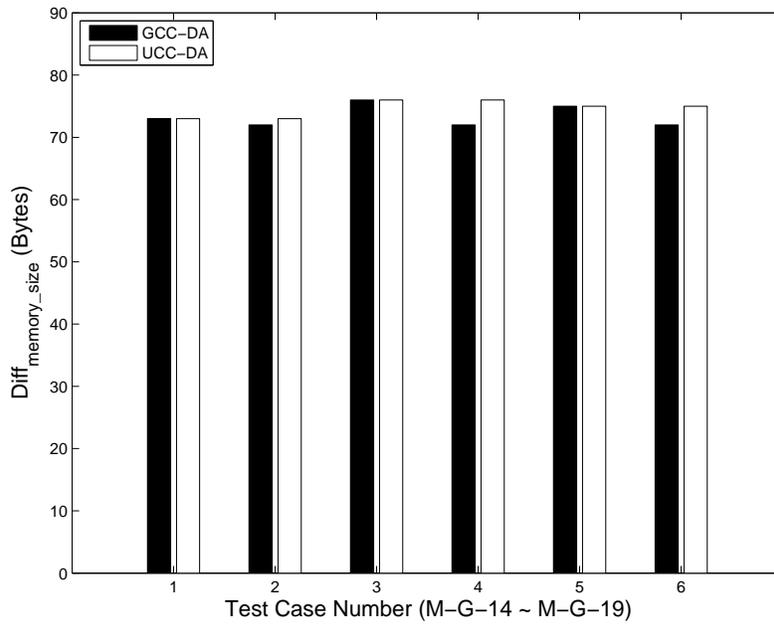


Figure 57: Worst-case stack size comparison between UCC-DA and GCC-DA.

memory space needed by the inserted variables. The common reason is that real-world code updates are fixing existing bugs and adding new features to handle unexpected conditions, which are likely to change and add new code rather than delete code.

6.2.2.5 Tradeoff between wasted space and binary differences

As shown in Figure 55, setting `SpaceT` to be 5B gives 56% script size reduction. I studied the impact on script size when reducing `SpaceT`. Intuitively, reducing `SpaceT` requires more variable relocations, and thus results in more code changes and larger script size.

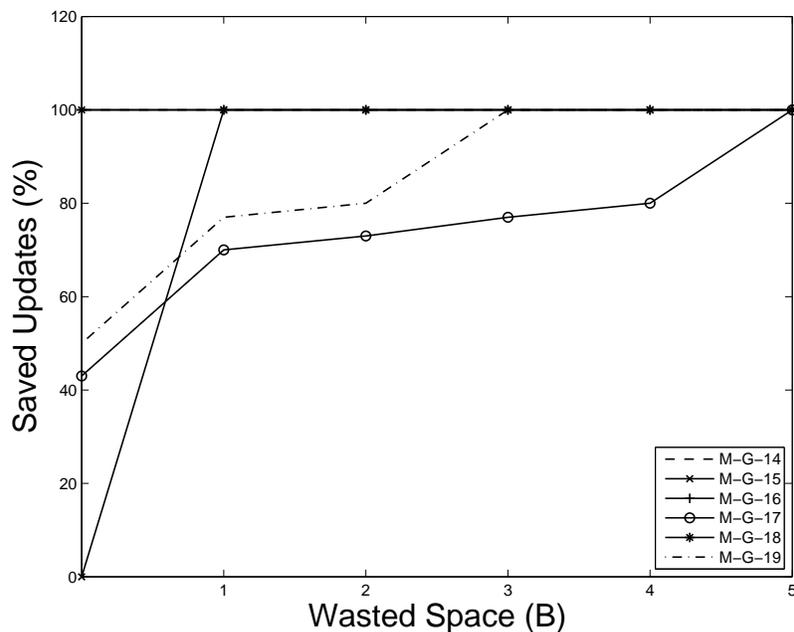


Figure 58: Tradeoff between the worst-case stack size and the instruction updates.

Figure 58 plots the relative script size under different threshold values. The results are normalized to the one when `SpaceT` is 5. The figure shows that the saved update percentage is higher with a larger `SpaceT`. In one case (M-G-15), I did not observe any savings if `SpaceT=0`, that is, the script size is the same as the one from GCC-DA.

From the figure, we can also find out that `SpaceT` does not need to be large to tolerate the instruction updates caused by data relocation. The worst-case stack size of the AES application is 67 Bytes for the given test cases. `SpaceT=5` is good enough to explore all

opportunities. Assuming the trend applies, I expect `SpaceT` to be larger for a real-world program, but it should still be within an acceptable range.

An interesting observation is that even setting the threshold `SpaceT` to be 0 can show improvement over the GCC-DA scheme. For example, test cases M-G-14, M-G-16, and M-G-18 achieve 100% update reduction when `SpaceT` is set to 0. This happens when the memory space occupied by the deleted variables is smaller than the space occupied by the inserted variables. Without the update-conscious scheme, the variables are ordered by the declaration sequence on stack. This may cause address shift to those unchanged variables that are declared after these new variables. However, using the update-conscious scheme, the new variables are first used to fill up the memory holes, and the extra new variables are always allocated on top of the unchanged ones. Thus, these unchanged variables will not be relocated, so that fewer instruction updates will be caused.

6.2.3 Updating general-purpose applications using UCC-RA and UCC-DA

6.2.3.1 Performance evaluation using `man-bench`

The experimental results in Section 6.2.1 and Section 6.2.2 show that using update-conscious register allocation and data allocation individually can reduce the update script sizes by 71% and 56% representatively. However, the performance loss caused by the update-conscious compilation schemes is very small, i.e., increasing the number of instructions in each execution by 4.7% and RAM usage by 1.8%.

Applying both the UCC-RA and UCC-DA schemes should combine the benefits and reduce the script size even more. I implemented the integration algorithm proposed in Chapter 3.2.3 and ran the integration algorithm on the `man-bench` M-G-14 ~ M-G-19. The maximum wasted space threshold is set to be 5 bytes. The generated script size comparison is shown in Figure 59. From the figure, the integrated scheme produces the smallest scripts compared to the individual UCC-RA and UCC-DA schemes.

For these test cases, using the integrated scheme does not introduce any extra runtime overhead. The reason is that the tested applications are simple and have no register pressure; the UCC-RA scheme can always find free registers to store the variables that are

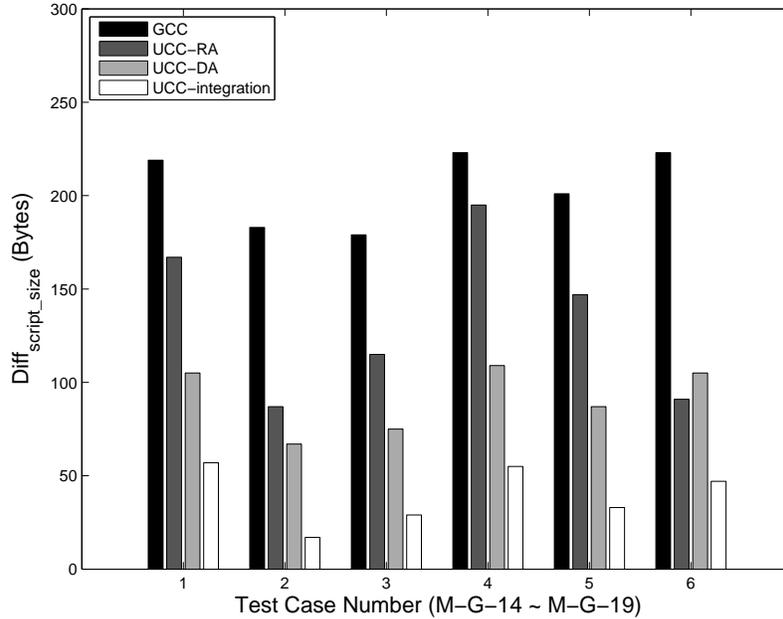


Figure 59: Script size comparison between the integrated scheme and the baseline scheme.

not tagged with a preferred register. There, UCC-RA does not introduce any extra move or spill instructions.

6.2.3.2 Performance evaluation using real-bench

Next, I used the real general-purpose benchmarks (R-G-1 ~ R-G-6) listed in Figure 44 to study the performance tradeoffs and energy savings for real software update cases. I used both GCC and the UCC designed for general-purpose applications to get new binaries after each update, and then generate the update scripts using instruction-based primitives.

Figure 60 shows the comparison results that list the number of script instructions per each primitive type, and the final script size (bytes) for these two compilation techniques. UCC did not add new instructions in these test cases. Therefore, the code generated from UCC has the same performance as that from GCC.

The average script size reduction for all the six test cases is 55% compared to GCC. This is because UCC reduces the instructions that need to be updated, thus the number of update

Case #	GCC Script Size (bytes)	#A	#R	#P	#C	#L	UCC Script Size (bytes)	#A	#R	#P	#C	#L
R-G-1	249	4	3	22	30	0	5	1	0	0	2	0
R-G-2	557	6	3	52	62	0	191	8	3	1	4	0
R-G-3	447	2	1	39	43	0	12	3	3	0	4	0
R-G-4	605	1	1	4	7	0	512	6	0	1	8	0
R-G-5	277	0	4	31	36	0	35	3	1	0	3	0
R-G-6	3981	12	6	143	162	0	1069	2	2	1	6	2

Figure 60: Script size comparison for real general-purpose updates (#A: add primitive; #R: remove primitive; #P: replace primitive; #C: copy primitive; #L: clone primitive).

primitives in the script is reduced. In addition, I 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 R-G-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.

Notice that the `clone` primitive is not frequently used in the script. This is because when the number of the instructions that can be “cloned” from the original code is not big enough, using the `replace` primitive is more efficient; for example, if there are N instructions that can be “cloned” from the original code, and the number of mapped registers is M . To update the code, I can choose either the `clone` primitive or `replace` primitive. The overhead of using each primitive is shown in the following equations.

$$Cost_{clone} = 5 + 2 * M \quad (6.3)$$

$$Cost_{replace} = 1 + 2 * N \quad (6.4)$$

$$Cost_{clone} < Cost_{replace} \Rightarrow N - M > 2 \quad (6.5)$$

As shown in Equation 6.5, only when $N - M > 2$ is satisfied, can I gain more benefit by choosing the `clone` primitive.

6.2.4 Updating DSP applications

In this section, I will evaluate the impact of the update-conscious compilation data allocation scheme (UCC-DA) for DSP applications and the CSOA/CGOA schemes. I will use both `man-bench` (M-D-1 ~ M-D-5) and `auto-bench` to study the performance tradeoffs.

6.2.4.1 Settings

To evaluate the proposed update-conscious ICSOA/ICGOA algorithms, I chose the Lance Compiler[44] to convert source code (C code) to intermediate representations (IRs) from which the access sequence and interference graph are extracted. I selected the DSPstone[21] benchmark suite that is widely used to measure the performance of DSP compilers. I adopted CSOA-Offsetstone[58] as the baseline and implemented ICSOA on top of it.

6.2.4.2 Script size comparison using `man-bench`

Single offset assignment. Figure 61 compares the software update overhead for CSOA and ICSOA. I used three script formats to do the comparison.

- *Simple code update script* that uses only the simple instruction-based primitives;
- *Advanced code update script* that uses all types of the instruction-based primitives;
- *Context-aware update script* that uses both instruction-based and data-based primitives.

Using the same script generator with ICSOA, the script size can be reduced by 32%. This is because the update-conscious strategy follows the variable coalesces and offset assignment of the old code. The generated code has better code similarity to the old version in terms of both offset assignment and instruction addressing mode. In test case M-D-1, the code update is very small such that the difference between the old and new offset assignments is not big. I did not see much benefit using ICSOA over CSOA.

When comparing different script generators, I observed that the *advanced* script generator produces a smaller script due to its usage of the `insert_access` primitive. When there is no variable access insertion but removal or update in the code update, the two script generators produce the same script, i.e., test cases M-D-4 and M-D-5.

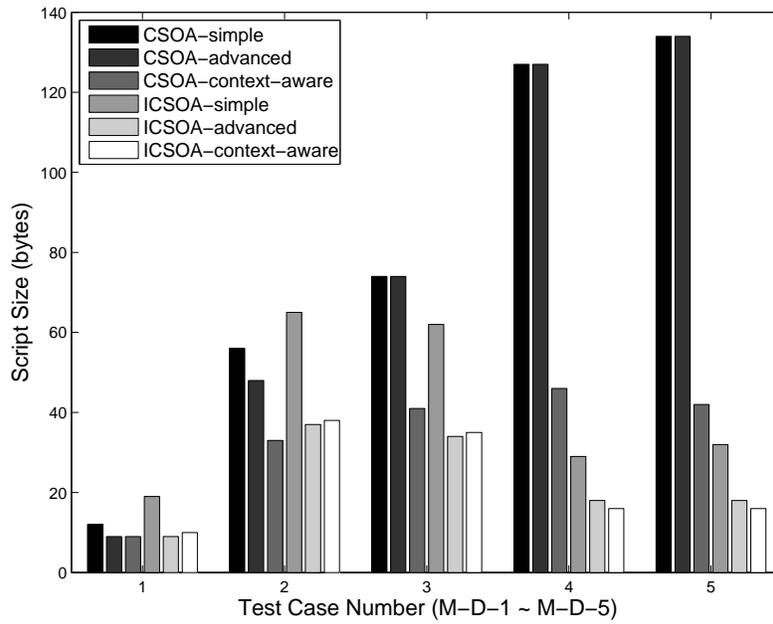


Figure 61: Script size comparison between ICSOA and CSOA (#addr register=1).

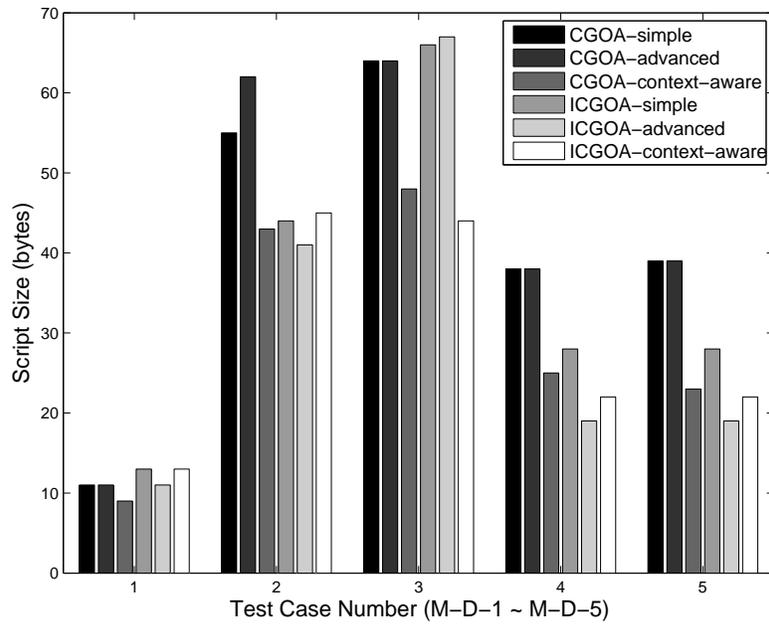


Figure 62: Script size comparison ICGOA and CGOA (#addr register=2).

The *context-aware* script generator produces smaller scripts when the code update is medium. Instead of sending individual instruction differences, it just sends out the data allocation differences, from which each node generates the new binary by itself, i.e., test cases M-D-4 and M-D-5. A significant script size reduction was observed by using this scheme. Adopting *context-aware* script tends to incur large complexity, i.e., test cases M-D-1 and M-D-3 where there is a small increase in script size due to the complexity required to specify the offset assignment change.

General offset assignment. When there are multiple ARs, Figure 62 compares CGOA and ICGOA schemes with the different script generators.

When there are more ARs, recompiling the program results in large changes in both the variable partition and offset assignment. For test case M-D-3, CGOA with a *context-aware* script has a larger size than with the *simple* script. This is due to the significant variable partition change, which requires more primitives to specify the new offset layout.

In conclusion, ICSOA/ICGOA is preferred when there are medium changes, while re-compilation is preferred when the change is small or large.

6.2.4.3 Code quality comparison using man-bench

Single offset assignment. As shown in Figure 63, ICSOA produces a number of instructions similar to CSOA. On average, the binary generated by ICSOA is 10% larger than the binary generated by CSOA. For the worst test case, i.e., test case M-D-3, the binary generated by ICSOA is 23% larger than CSOA. Because the ICSOA scheme does the data allocation incrementally based on the *coalesced access graph* of the old version, the old variable coalescing result is retained in the new version to improve code similarity. As a result, the code generated by ICSOA is not as efficient.

To better understand the code quality difference between the two approaches, Figure 64 shows the breakdown of the execution overhead. I separated the new code at the intermediate representation (IR) level into the changed and unchanged parts. I then created their mapping to the binary level code segments.

When the offset assignment is changed, the same IR instruction may be translated to

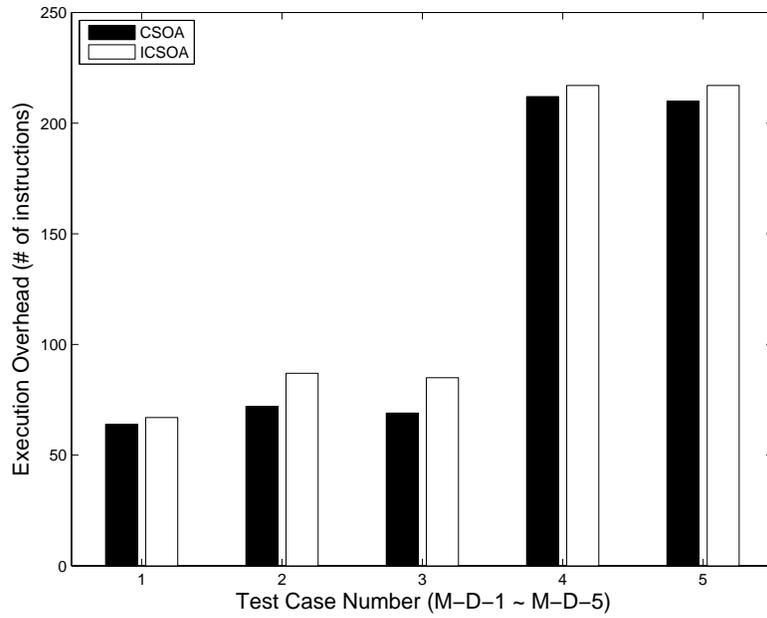


Figure 63: Code quality comparison between CSOA and ICSOA.

Test Case#	CSOA				ICSOA			
	T1	T2	T3	T4	T1	T2	T3	T4
M-D-1	0	7	1	0	0	7	2	0
M-D-2	1	7	9	0	0	8	12	3
M-D-3	1	7	7	0	0	10	12	6
M-D-4	4	24	0	0	0	24	2	1
M-D-5	4	22	0	0	0	24	2	1

Figure 64: Execution overhead breakdown.

different binary instructions in the old and new code. The binary code differences could be categorized as two types: (1) updating the addressing mode of the related binary instructions, such as the first memory access in Figure 25; (2) adding addressing mode change instructions. The first type does not change the number of instructions, as no extra instruction is added, but for the second type one extra instruction is added per change. When studying the code quality, I divided the overhead into four categories as follows. T1-T3 show how efficient the offset assignment algorithm is, and T4 shows how the extra patch affects the final result.

- T1: AR loading instructions removed from the old code;
- T2: AR loading instructions inserted into the old code;
- T3: AR loading instructions inserted into the new code;
- T4: ALU instructions inserted into the new code.

Comparing columns T1 and T2 of both CSOA and ICSOA in Figure 64, we can see that CSOA generates fewer binary instructions for the unchanged IR part. It removes more AR loading instructions, and inserts fewer such instructions. For the new code part, CSOA generates fewer AR loading instructions. When performing complete recompilation, CSOA uses the new access sequences and variable interferences of the whole function, and thus can generate a better offset assignment.

Column T4 shows the number of ALU instructions generated by compiling the new assembly code. Since ICSOA needs to add patch variables to remove the interferences due to overlapped live ranges, it adds several `mov` instructions in the code, which produces more T4 type instructions.

General offset assignment. For the test case M-D-3 that has the largest code quality difference, I increased the number of available ARs, and found that with more available ARs the code quality difference is reduced, as shown in Figure 65. The extra instruction number drops from 20% to 6% when the address register number is increased from 1 to 4. This is because with more ARs, the variables are partitioned into smaller sets. The software update tends to create less new interference and needs fewer patch variables. Fewer interferences result in less overhead in ICSOA.

The update-conscious data allocation scheme trades the run-time code performance for

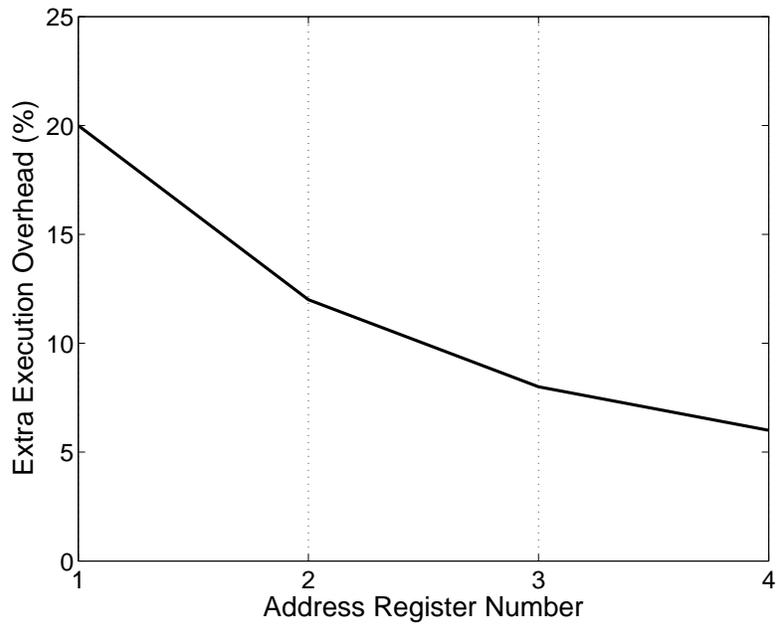


Figure 65: Code quality comparison between ICGOA and CGOA.

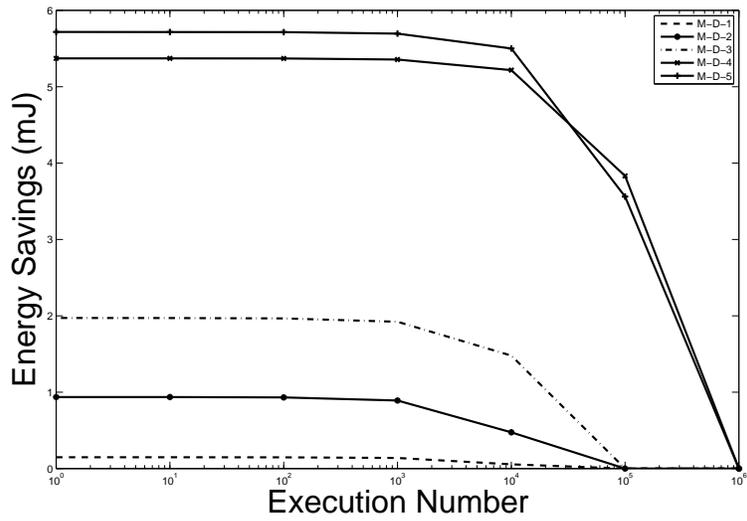


Figure 66: The energy savings for DSP applications.

the transmission overhead during software update. Thus, the overall energy savings depend on the number of times the target binary will execute before retiring. Figure 66 shows the energy savings of ICSOA over CSOA as a function of Cnt , which is projected from the execution profiles and the code update frequency.

From Figure 66, with the increase of the execution number of one application, the overall energy savings that are achieved by using the update-conscious compilation scheme are reduced. This is because the energy saved in the one-time binary transmission is overwhelmed by the extra run-time overhead. When a large Cnt is predicted, the update-conscious compilation will fall back to the CSOA scheme in order to achieve overall energy efficiency.

6.2.4.4 Performance evaluation using auto-bench

I next used `auto-bench` to compare the script size and performance between CSOA and ICSOA schemes. I inserted changes randomly into a file (`verify.c`) to study the robustness of my proposed scheme. The inserted code involves the use of both existing and new variables. The ratio of these two types is 1:1, and the sizes of the inserted/changed code range from 5% to 60% of the original code. Given an update percentage, I randomly generated 500 test cases and reported the average.

The script size comparison is shown in Figure 67. For all three types of script generation schemes, ICSOA reduces more of the update script size and thus the software update transmission overhead. However, the results show that ICSOA achieved the maximum script size reduction when the update percentage is between 10% and 40%. This is because ICSOA benefits more when most of the update is caused by the data allocation changes rather than new/updated instruction operations. When the update percentage is too big, i.e., larger than 40%, most changes are new or updated instructions. When the update percentage is too small, i.e., smaller than 20%, the data allocation table is less likely to change even with recompilation. Thus, the benefits from ICSOA are limited.

The code quality is compared in Figure 68. Larger code update percentage, i.e. over 40%, has more live range extension of old variables, which produces more patch variables and instructions. Thus, the code produced by the recompilation scheme has a larger number

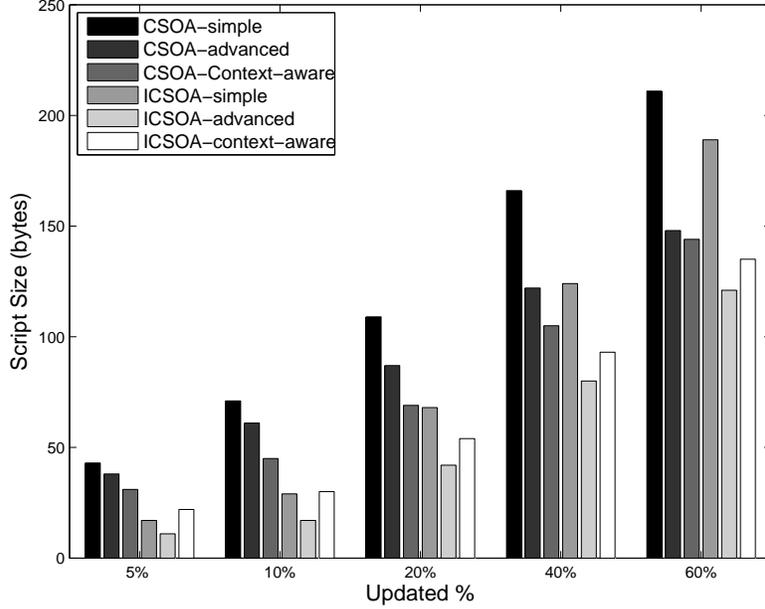


Figure 67: Script size comparison using auto-bench.

of T4 type instructions, and the code generated by the ICSOA scheme has a worse execution performance.

From Figure 67 and Figure 68, we can see that when the code update percentage is between 10% and 20%, using the update-conscious offset assignment scheme can save about 30% of the transmission overhead, assuming that the advanced script is used, with about 4% extra runtime overhead.

From the experimental results, we can also see that using instruction-based primitives works better with incremental compilation (ICSOA), and using data-based primitives (i.e., context-aware primitives) works better with full recompilation (CSOA). This is because context-aware primitives trade individual updates for setting up the auxiliary data structures and letting the sensors construct these updates. Recompiling the new code changes the data allocation and thus has a relatively greater number of instruction changes, which enables context-aware primitives, gaining more benefits. When the code is generated from incremental compilation, the saving is not large enough to balance the overhead in setting

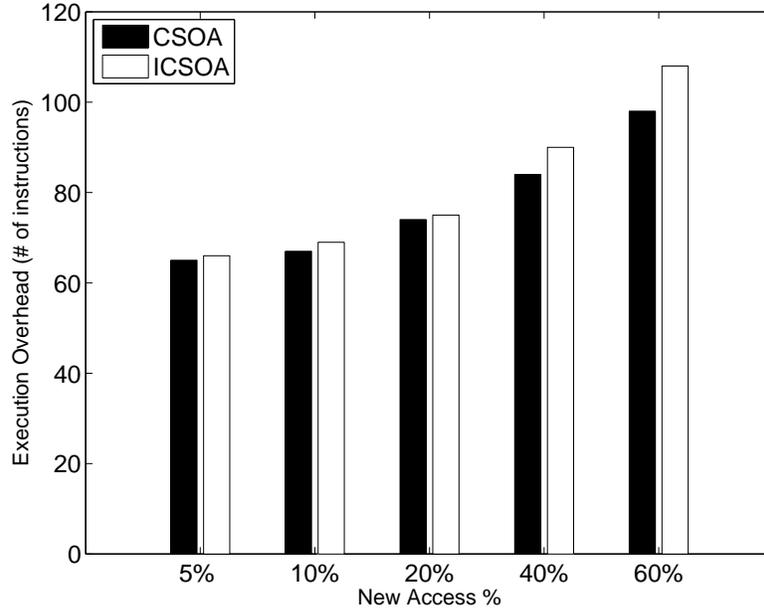


Figure 68: Code quality comparison using auto-bench.

up the data structures. In this case, it is beneficial to simply use the instruction-based primitives.

6.2.4.5 Performance evaluation using real-bench

In this section, I used the real DSP test cases (R-D-1 ~ R-D-3) to study the script size and performance tradeoff for overall energy efficiency. Test cases R-D-1 and R-D-2 are medium-level updates, while R-D-3 is a high-level update. In the experiments, I compared the update script size, generated binary performance, and long-term energy savings between the baseline CSOA and the proposed ICSOA schemes. The update scripts are then generated using different update primitives described previously.

Figure 69 compares the generated script size using different techniques. When comparing CSOA-simple and ICSOA-simple, we can see that update-conscious data allocation (ICSOA) produces the binary that is more similar to the old binary when the update level is relatively low. When the code has significant changes, e.g., test case R-D-3 introduces 32% code

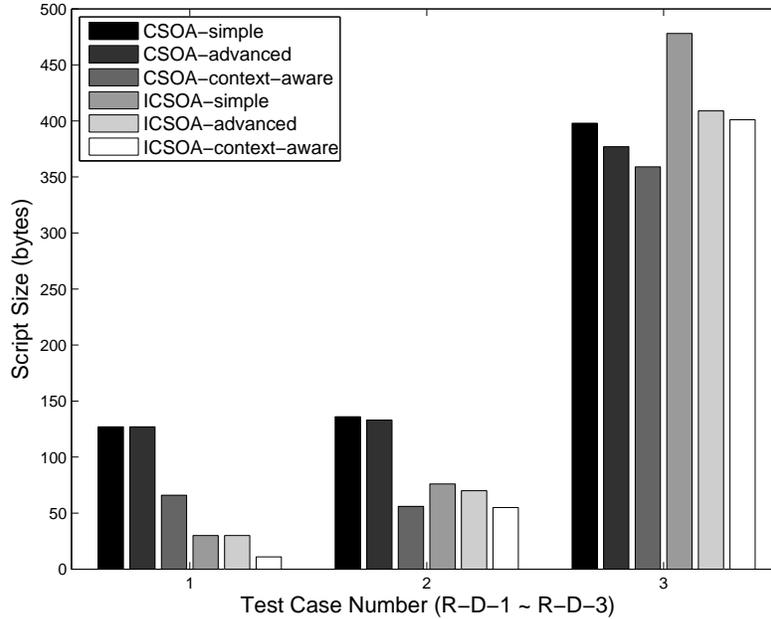


Figure 69: Script size comparison between CSOA and ICSOA (#addr register=1).

changes, the old and new code segments are mixed, such that the benefit from keeping the old data offset assignment diminishes.

Using advanced primitives helps to further reduce the script size. The advanced primitives tend to take more effect when the code update level is higher. This is because they work under certain circumstances, e.g., only when the update involves inline functions, the `clone` primitive can help to reduce the script size, and higher-level updates provide more opportunities for these primitives to take effect.

Using data-based primitives achieves more script size reductions than using instruction-based primitives. Although the former increases the code regeneration overhead on the sensors, it helps reduce the script size significantly, especially when the data allocation update is significant. For example, compared to CSOA, ICSOA produces a data allocation that is more similar to the old version. Due to the amortized setting overhead of data-based primitives, CSOA benefits more than ICSOA when adopting these primitives.

Figure 70 shows the experimental results when there are two address registers. Without considering the old binary, a full recompilation often causes more data allocation differences

when the update is at a small or medium level.

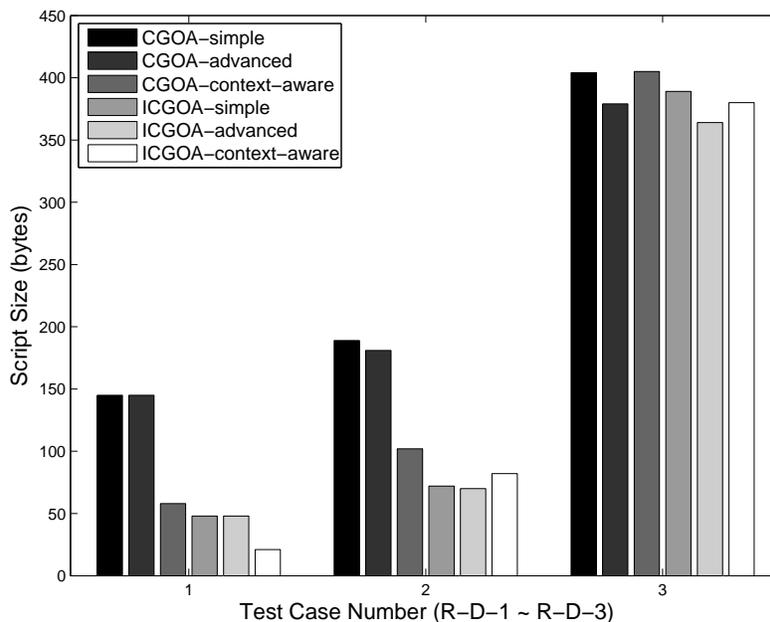


Figure 70: Script size comparison between CGOA and ICGOA (#addr register=2).

Figure 71 compares the generated code performance between CSOA and ICSOA. When the code changes, inheriting the old data allocation result may not result in efficient code. Thus, ICSOA generates more instructions compared to CSOA. For these real test cases, on average, ICSOA increases the run-time overhead by 3.7%.

6.3 SOFTWARE UPDATE STRATEGY IN SA-WSN

From the thorough analyses of the experimental data in the preceding section, we can see that an update-conscious compilation strategy is preferred for small to medium changes, while its benefits diminish when the update becomes large. Due to the nature of software updates, it is less likely to find one threshold value (e.g., the percentage of changed instructions) that can unambiguously determine if UCC should be applied. For example, given two applications that both have been updated by 30% of their code, the one that has the updates concentrated in one function should apply UCC, while the one that has the updates scattered around

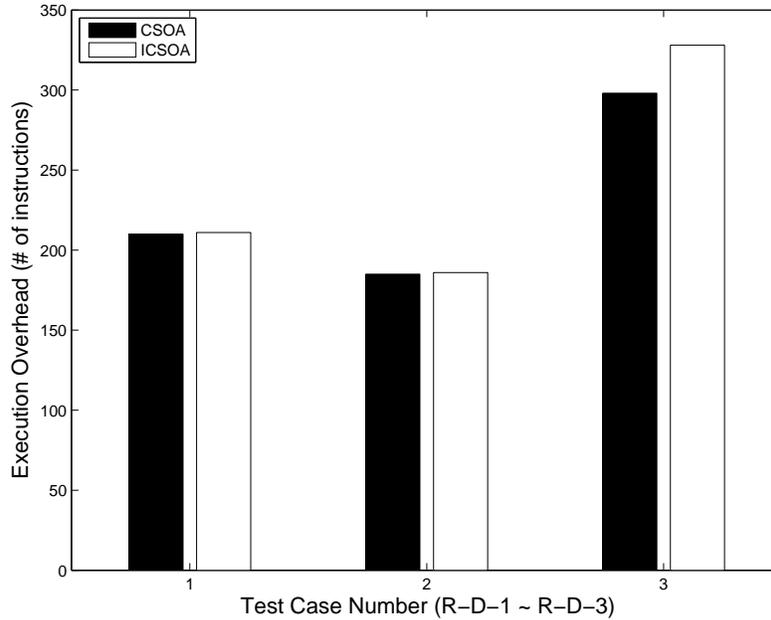


Figure 71: Code quality comparison between CSOA and ICSOA (#addr register=1).

should not. Even for the updates in one function, the one that has the updates concentrated within a small code range should apply UCC, while the one that has the updates scattered around the function should not.

When UCC is applied as the last optimization pass, the result from UCC is always the best (see the analyses in Section 3.1). However, when UCC is integrated in a different place in the compilation, it is worth studying its interaction with other optimization passes.

6.4 PATCH DISSEMINATION WITHIN MA-WSNS

I then simulated the patch dissemination in an MA-WSN and compared the message overhead and dissemination time using different code distribution protocols.

6.4.1 Settings

I implemented MCP on the TinyOS [76] platform. For comparison, I also implemented Melete [83] and studied various network settings using TOSSIM [48]. I simulated mesh MA-WSNs of different sizes. I set the default spacing factor to 15 and modeled the lossy communication channel using the tool provided by TinyOS. There are four applications each of which is uniformly distributed across the network. In the default setting, 30% of the sensors have application A and there is a request from the sink to reprogram 20% of the other sensors to run A. MCP disseminates the code from in-network sources instead of from the sink.

6.4.2 Message overhead

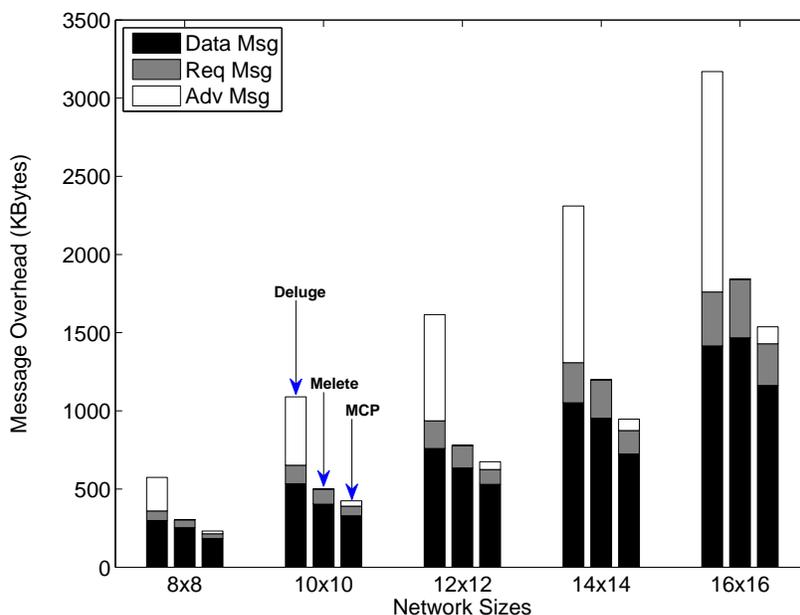


Figure 72: Message overhead.

Figure 72 shows the breakdown of the number of messages with different dissemination protocols. Without considering advertisement messages, Melete and Deluge have about the same message overhead, which was also reported in [83]. There are a large number of ADV messages in Deluge, and a negligible number in Melete. The reason for such difference

is that Deluge depends heavily on incoming ADV messages, e.g., a sensor node only sends out new requests if it receives ADV messages indicating its neighbors have more up-to-date data. In Melete, requesters receive the command from the sink code and then know the target application and its size. The requesters can proactively send out more requests after timeouts or after receiving one complete page. The ADV messages contribute to 37-40% of the total message overhead in Deluge.

My scheme takes an approach similar to Melete but requires some ADV messages to update the AIT before, during, and after the code switch. The ADV's overhead is low compared to the request and data transfer message overhead. On average, my scheme reduces the message overhead from Melete by about 20%. The main reason for this reduction is that Melete tends to have multiple responders within a small range and has a higher possibility of signal collision. MCP alleviates the problem by choosing one nearby source, which reduces the number of data packets in transmission.

6.4.3 Completion time

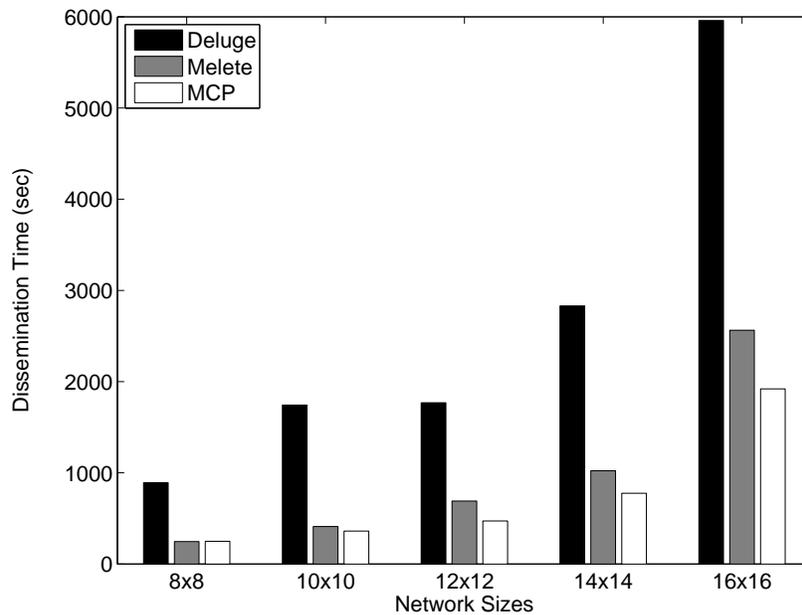


Figure 73: Dissemination time.

Figure 73 compares the dissemination completion time of the different protocols. For the

Deluge result, I recorded the time interval used by all requesters to complete the downloading of new code. In practice, the Deluge protocol may still proceed to flood all sensors, since it is not designed to update a subset of sensors. MCP requires less time to finish dissemination; on average, it saves 45% and 25% over Deluge and Melete respectively.

6.4.4 Sensitivity to node distribution

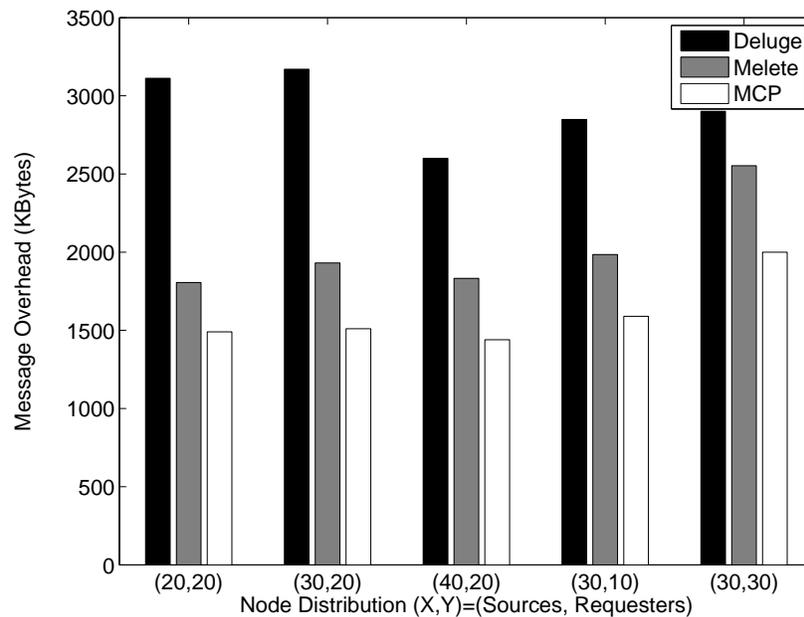


Figure 74: Dissemination with different numbers of sources and requesters.

Figure 74 illustrates message overhead with a different number of sources and requesters. I omit the dissemination time figure which exhibits similar results. Along the X axis, (a,b) denotes that out of all the sensor nodes, a% sources and b% requesters are randomly selected in the field. I observed that the overhead tends to increase with more requesters and fewer sources. The difference is not significant.

Figure 75 compares the message overhead when sources and requesters are distributed with location concentration. **EvenD** denotes that all nodes are evenly distributed. **CornerD** denotes that sources and requesters are distributed at the two diagonal corners of the rectangle field. **SideD** denotes that sources and requesters are distributed along two sides of the

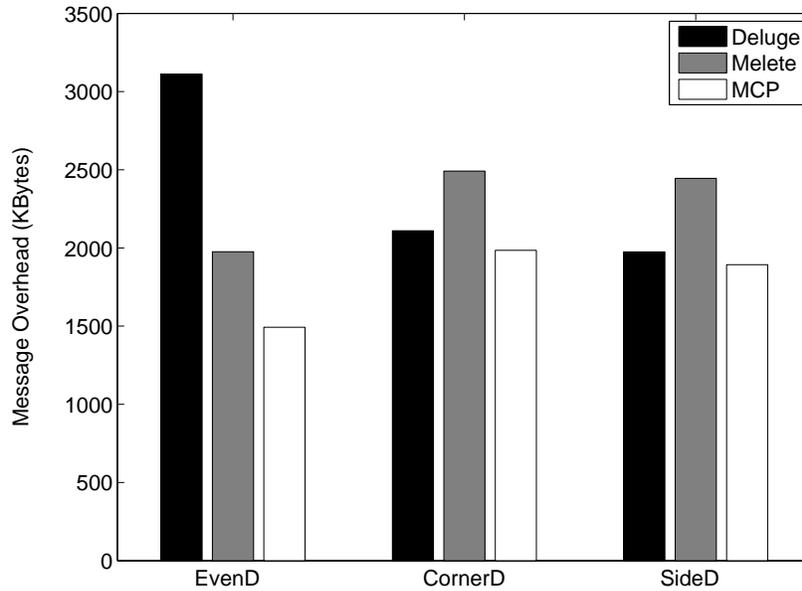


Figure 75: Dissemination with uneven source/requester node distribution.

field. From the figure, Melete has better performance than Deluge under even distribution. However, it generates significant conflicts and performs worse than Deluge when the nodes are unevenly deployed. MCP has consistently better results over Melete and Deluge. For the corner and side settings, MCP and Deluge are similar, as almost all nodes are involved in the dissemination.

6.4.5 Sensitivity to application sizes

Figure 76 shows message overhead with different application sizes. Due to the epidemic dissemination, Deluge exhibits approximately linear message overhead when increasing the application size from 8 to 16 pages. Both Melete and MCP greatly reduce the communication overhead; however, they have slightly more than linear message overhead due to independent page requesting from requesters. MCP has a nearly constant message overhead reduction versus Melete, varying from 17.5% for 8 pages to 18.1% for 16 pages.

6.4.6 Sensitivity to cache sizes

Figure 77 summarizes the message overhead of Melete and MCP with different cache sizes, i.e., the number of code pages that may be cached in memory. Here $N=1$ denotes that

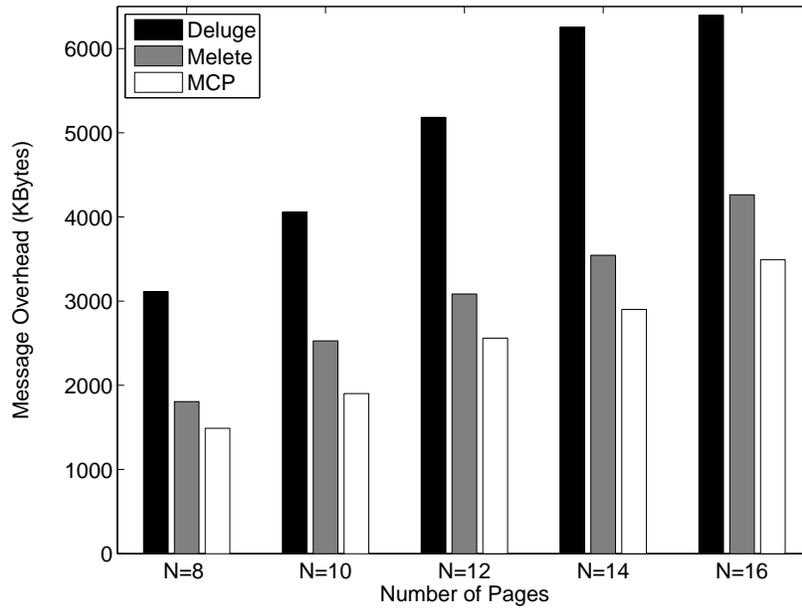


Figure 76: Dissemination with different number of pages.

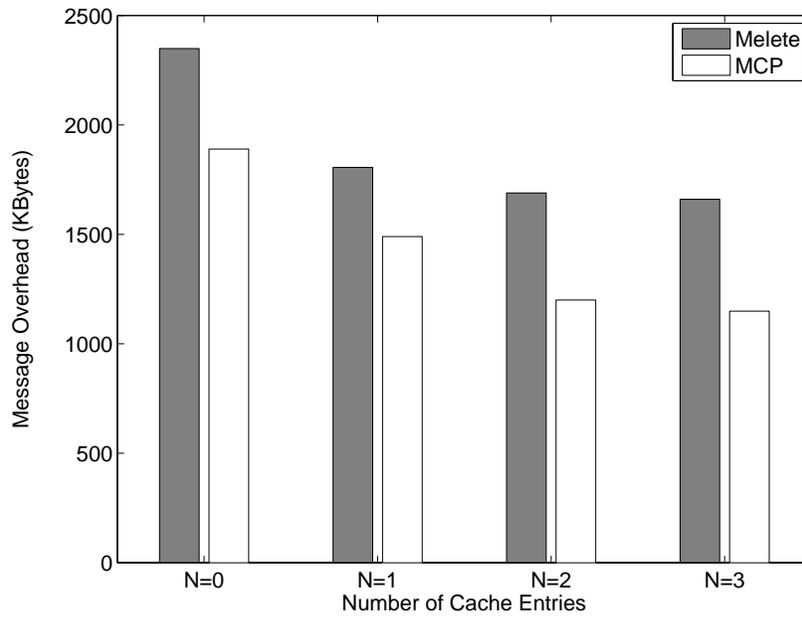


Figure 77: Dissemination with Different Cache Sizes.

there is no caching. From the figure, MCP achieves significant reduction in communication overhead when caching one or two future pages, and diminishing benefits with larger cache sizes. The reason is that in MCP, a request message can preempt a working node (a source, a requester, or a forwarder) if that node works on a page with a larger page number, and the page index difference is larger than one. In this way, MCP prioritizes slow requesters such that they can keep up the pace with the nearby dissemination and take advantage of cached packets.

7.0 FUTURE DIRECTIONS AND CONCLUSION

This chapter addresses future research directions and concludes the dissertation.

7.1 FUTURE WORK

Although the designed software update framework has achieved the design goal and gained significant energy savings for the WSN platform, there are several directions that are worth exploring in the future.

7.1.1 Apply to different platforms

One future research direction is to apply the techniques proposed in this research to other computing platforms and evaluate the benefits to be gained compared to the traditional solutions.

The software update management framework can be adapted to the smart phone network. The popularity of smart phones has stimulated interest in developing and using applications. By September 1, 2010, there were over 250,000 applications for the iPhone [18] platform. According to Tech Crunchies [20], the average number of applications installed on an iPhone is 65. Because of the rapidly growing demand, and the fast pace of development, these applications tend to be updated very frequently. Multiple releases of one application could be launched in a month. Efficiently updating these applications could be an issue, since frequent software updates can deplete the energy stored in the battery and consume too much bandwidth to satisfy the QoS of the other running applications.

Applying update-conscious compilation and differential patching techniques to these smart phone platforms can reduce the number of bytes that need to be transmitted to the phones. This will reduce not only the update time but also the data usage. With the multicast-based code dissemination protocol installed, the smart phones will be able to download new applications from the peer phones via Bluetooth, which will reduce data usage and decrease the impact on other running applications that heavily use the network.

7.1.2 Other update-conscious compilation schemes

In this research, I focused on optimizing the register allocation and data allocation to improve the similarity between different binary versions. Aside from this, other UCC research opportunities exist, such as UCC instruction selection and UCC instruction scheduling.

Instruction selection transforms the mid-level intermediate representation (IR) into a low-level IR that is very close to its final target language. The traditional “tile covering” algorithms try to optimize the run-time overhead while selecting the proper “tiles” to cover the IR tree parts with the least cost. Each tile represents one instruction type that is available on the target machine architecture. The UCC instruction selection algorithm should take the instruction selection results of the old version as input while generating the new version and consider not only run-time overhead but also code similarity. This trade-off between run-time overhead and transmission overhead can be studied.

The functional update primitive design favors continuous updates, because it allows the use of one primitive to describe multiple updated instructions. Comparing two updates that have the same number of new instructions, where one has all the updates concentrated at one or two update points, and the other has all the updates scattered in the existing code, the first will have a smaller update script. Thus, while doing instruction scheduling, if we can advance or delay certain instructions to implode the updates, we will be able to reduce the patch size. This may affect run-time performance by introducing more “stalls”, and this trade-off needs to be studied in future research.

7.2 CONCLUSION

Wireless sensor networks (WSNs) have been widely used. The software running on the deployed sensors needs to be updated for various reasons. Since sensors are usually left unattended after deployment, the update must be done by energy-expensive wireless communication. Experimental results have shown that the energy spent transmitting one bit one hop is equivalent to the energy consumed by executing 10 instructions. When updates are frequent, consuming too much energy for software updates may greatly shorten the lifetime of the network.

In this dissertation, I designed a software update management framework that addresses the challenges using three integrated components.

The update-conscious compiler (UCC) improves binary code similarity by generating a new binary that is similar to the old version. In this dissertation, I developed UCC register allocation and UCC data allocation schemes. The update-conscious register allocation (UCC-RA) scheme formulates the problem as an ILP problem. The objective is to minimize the overall energy consumption including the run-time overhead in terms of the number of “load”, “store” and “move” instructions, and the software update overhead in terms of the binary difference from the old version. The update-conscious data allocation scheme for general-purpose applications uses a threshold-based solution to minimize the variable relocation under a certain memory usage constraint. The update-conscious data allocation scheme for DSP applications integrates the binary similarity in the existing CSOA and CGOA algorithms, which generates the new binary with similar data allocation, resulting in similar addressing mode selections for the memory access instructions. UCC strives to achieve overall energy efficiency during compilation by considering the number of invocations the new binary will have, the memory space constraint of the target platform, and the memory usage of the new binary.

The generated new binary is then compared with the old binary to generate an update patch. In order to further reduce the patch size, several sets of patch primitives are designed. The `simple` update primitives directly summarize the lower-level binary differences in the patch. It is easy to interpret at the sensor side, but may result in a larger patch. The

advanced instruction-based primitives summarize the root cause that affects more than one instruction in one primitive. The data-based primitives are used to describe the data allocation changes and the affected addressing mode changes. The latter two types help reduce patch size, but require a more powerful interpreter at the sensor side.

A multicast-based stateful code distribution protocol (MCP) is then used to disseminate the generated binary to the sensors. This protocol stores the routing information of nearby source nodes on individual sensors. When there is a need to update the code, the requests can be directed to the sources without flooding the network. The memory on the sensors is wisely divided into cache packages received from different sources.

My framework integrates the three components to address the critical software update problem in WSNs. The experimental results showed that when they work cooperatively together, my software update management framework can greatly reduce the number of bytes sent across the network, shorten the time required to disseminate the update, and achieve overall energy efficiency.

8.0 BIBLIOGRAPHY

- [1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools, second edition*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.
- [2] A. W. Appel and L. George, “Optimal spilling for cisc machines with few registers,” In *Proceedings of the ACM SIGPLAN 2001 conference on programming language design and implementation (PLDI)*, pages 243–253, 2000.
- [3] Atmel atmega128L reference manual.
<http://www.atmel.com/atmel/acrobat/doc2467.pdf>.
- [4] S. Atri, J. Ramanujam, and M. T. Kandemir, “Improving offset assignment on embedded processors using transformations,” In *Proceedings of the 7th international conference on high performance computing (HiPC)*, pages 367–374, 2000.
- [5] K. C. Barr and K. Asanović, “Energy-aware lossless data compression,” In *ACM Transactions on Computer Systems (TOCS)*, 24(3):250–291, 2006.
- [6] D. H. Bartley, “Optimizing stack frame accesses for processors with restricted addressing modes,” In *Software practice and Experience*, 22(2):101–110, 1992.
- [7] S. Bhattacharya, A. Saifullah, C. Lu, G.C. Roman, “Multi-Application Deployment in Shared Sensor Networks Based on Quality of Monitoring,” In *IEEE the 16th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2010.
- [8] M. Berkelaar *et al.*, Lp_solve 5.5, <http://lpsolve.sourceforge.net/5.5/>.
- [9] M. P. Bivens and M. L. Soffa, “Incremental register reallocation,” in *Software practice and experience*, 20(10):1015–1047, 1990.
- [10] P. Bonami, L.T. Biegler, A.R. Coon, G. Cornuejols, I.E. Grossmann, C.D. Laird, J. Lee, A. Lodi, F. Margot, N. Sawaya, A. Wachter, “An algorithmic framework for convex mixed integer nonlinear programs,” IBM Research Report, 2005.
- [11] P. Briggs, K. D. Cooper, and L. Torczon, “Improvements to graph coloring register allocation,” In *ACM transactions on program languages and systems (TOPLAS)*, 16(3):428–455, 1994.
- [12] E.H. Callaway, Jr, *Wireless Sensor Networks: Architectures and Protocols*, CRC Press, 2003.

- [13] G. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein, “Register allocation via coloring,” In *Computer languages*, 6:47–57, 1981.
- [14] T. M. Chilimbi, B. Davidson, and J. R. Larus, “Cache-conscious structure definition,” In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 13–24, 1999.
- [15] Y. Choi and T. Kim, “Address assignment combined with scheduling in DSP code generation,” In *Proceedings of the 39th conference on design automation (DAC)*, pages 225–230, 2002.
- [16] F. Chow and J. Hennessy, “Register allocation by priority-based coloring,” In *Proceedings of the SIGPLAN symposium on compiler construction*, volume 19, pages 222–232, 1984.
- [17] N. Coopriider and J. Regehr. “Pluggable abstract domains for analyzing embedded software,” In *Proceedings of conference on languages, compilers, and tools for embedded systems (LCTES)*, pages 44–53, 2006.
- [18] Apple Inc., *iPhone user guide*, 2010.
- [19] Crossbow, *Mica2/MicaZ/Imote2 Data Sheet*.
- [20] Tech Crunchies, “Internet statistics and numbers,” 2010. <http://www.techcrunchies.com>.
- [21] DSPStone benchmark suite, 1995.
<http://iss.rwth-aachen.de/Projekte/Tools/DSPSTONE>.
- [22] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt, “Run-time dynamic linking for reprogramming wireless sensor networks,” In *SenSys ’06: Proceedings of the 4th international conference on embedded networked sensor systems (SenSys)*, pages 15–28, 2006.
- [23] P. K. Dutta, J. W. Hui, D. C. Chu, and D. E. Culler, “Securing the deluge network programming system,” In *Proceedings of the 5th international conference on information processing in sensor networks (IPSN)*, pages 326–333, 2006.
- [24] M. Franz, “Adaptive compression of syntax trees and iterative dynamic code optimization: Two basic technologies for mobile-object systems,” In *J. Vitek and Ch. Tschudin (Eds.), Mobile Object Systems: Towards the Programmable Internet, Springer Lecture Notes in Computer Science, No. 1222*, 1997.
- [25] C. Fraser, “An instruction for direct interpretation of lz77-compressed programs,” Microsoft Technical Report, MSR-TR-2002-90, 2002.
- [26] C. Fu and K. Wilken, “A faster optimal register allocator,” In *Proceedings of the 35th annual ACM/IEEE international symposium on microarchitecture (MICRO)*, pages 245–256, 2002.
- [27] GCC, the GNU compiler collection. <http://gcc.gnu.org>.
- [28] L. George and A. W. Appel, “Iterated register coalescing,” In *ACM transactions on program languages and systems (TOPLAS)*, 18(3):300–324, 1996.

- [29] D. W. Goodwin and K. D. Wilken, "Optimal and near-optimal global register allocations using 0–1 integer programming," In *Software practice and experience*, 26(8):929–965, 1996.
- [30] W. R. Heinzelman, J. Kulik, and H. Balakrishnan, "Adaptive protocols for information dissemination in wireless sensor networks," In *Proceedings of the 5th annual ACM/IEEE international conference on mobile computing and networking (Mobicom)*, pages 174–185, 1999.
- [31] HP openview change and configuration management.
<http://www.managementsoftware.hp.com/solutions/ascm/index.html>.
- [32] J. W. Hui and D. Culler, "The dynamic behavior of a data dissemination protocol for network programming at scale," In *Proceedings of the 2nd international conference on eEmbedded networked sensor systems (SenSys)*, pages 81–94, New York, NY, 2004.
- [33] IBM tivoli configuration manager.
<http://ibm.com/software/tivoli/products/configmgr/>.
- [34] C. Jaramillo, R. Gupta, and M. L. Soffa, "Capturing the effects of code improving transformations," In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, page 118, Washington, DC, 1998.
- [35] J. Jeong and D. Culler, "Incremental network programming for wireless sensors," In *Proceedings of sensor and ad hoc communications and networks(SECON)*, pages 25–33, 2004.
- [36] P. Juang, H. Oki, Y. Wang, M. Martonosi, L.S. Peh, and D. Rubenstein, "Energy Efficient Computing for Wildlife Tracking: Design Tradeoffs and Early Experiences with ZebraNet," In *ACM/IEEE International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS)*, pages 96–107, 2002.
- [37] J.M. Kahn, R.H. Katz, and K.S.J. Pister, "Emerging Challenges: Mobile Networking for 'Smart Dust'," In *Journal of Communications and Networks*, 2(3):188–196, 2000.
- [38] D. R. Koes and S. C. Goldstein, "A global progressive register allocator," In *Proceedings of the 2006 ACM SIGPLAN conference on programming language design and implementation (PLDI)*, pages 204–215, 2006.
- [39] J. Koshy and R. Pandey, "Remote incremental linking for energy-efficient reprogramming of sensor networks," In *Proceedings of the 2nd European workshop on wireless sensor networks*, pages 354–365, 2005.
- [40] P. A. Kulkarni, D. B. Whalley, G. S. Tyson, and J. W. Davidson, "Practical exhaustive optimization phase order exploration and evaluation," In *ACM Transactions on Architecture and Code Optimization*, pages 6:1:1–1:36, April 2009.
- [41] P. Lanigan, R. Gandhi, and P. Narasimhan, "Sluice: secure dissemination of code updates in sensor networks," In *Proceedings of the 26th IEEE international conference on distributed computing Systems (ICDCS)*, pages 53–53, 2006.
- [42] P. Lapsley, J. Bier, E. A. Lee, and A. Shoham, *DSP Processor fundamentals: architectures and features*. Wiley-IEEE Press, 1996.

- [43] J. Lau, S. Schoenmackers, T. Sherwood, and B. Calder. “Reducing code size with echo instructions,” In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (LCTES)*, pages 84–94, 2003.
- [44] R. Leupers, “LANCE: a C compiler platform for embedded processors,” In *Embedded Systems*, 2001.
- [45] R. Leupers and F. David, “A uniform optimization technique for offset assignment problems,” In *Proceedings of the 11th international symposium on System synthesis (ISSS)*, pages 3–8, 1998.
- [46] R. Leupers and P. Marwedel, “Algorithms for address assignment in DSP code generation,” In *Proceedings of the IEEE/ACM international conference on computer-aided design*, pages 109–112, 1996.
- [47] P. Levis and D. Culler, “Maté: a tiny virtual machine for sensor networks,” In *Proceedings of the 10th international conference on architectural support for programming languages and operating systems (ASPLOS)*, pages 85–95, 2002.
- [48] P. Levis, N. Lee, M. Welsh, and D. Culler, “TOSSIM: accurate and scalable simulation of entire TinyOS applications,” In *Proceedings of the 1st international conference on Embedded networked sensor systems (SenSys)*, pages 126–137, 2003.
- [49] P. Levis, N. Patel, D. Culler, and S. Shenker, “Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks,” In *Proceedings of the 1st conference on symposium on networked systems design and implementation (NSDI)*, 2004.
- [50] W. Li, Y. Zhang, J. Yang, and J. Zheng, “UCC: update-conscious compilation for energy efficiency in wireless sensor networks,” In *Proceedings of the 2007 ACM SIGPLAN conference on programming language design and implementation (PLDI)*, pages 383–393, 2007.
- [51] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang, “Storage assignment to decrease code size,” In *ACM transactions on programming languages and systems (TOPLAS)*, 18(3):235–253, 1996.
- [52] P. J. Marrn, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, and K. Rothermel, “Flexcup: A flexible and efficient code update mechanism for sensor networks,” In *Proceedings of the 3rd european workshop on wireless sensor networks (EWSN)*, pages 212–227, 2006.
- [53] W.P. McCartney, and N. Sridhar, “Abstractions for Safe Concurrent Programming in Networked Embedded Systems,” In *Proceedings of the 1st international conference on Embedded networked sensor systems (SenSys)*, 2006.
- [54] Microsoft systems management server.
<http://www.microsoft.com/smsserver>.
- [55] E. Morel, and C. Renvoise, “Global optimization by suppression of partial redundancies,” In *Communications of the ACM*, 22(2), 1979.
- [56] R. Muth, S. Debray, S. Watterson, and K. D. Bosschere, “alto: A link-time optimizer for the compaq alpha,” In *Software Practice and Experience*, 31:67–101, 1999.

- [57] L. Nachman, J. Huang, J. Shahabdeen, R. Adler, and R. Kling, “Imote2: Serious computation at the edge,” In *International Wireless Communications and Mobile Computing Conference*, pages 1118–1123, 2008.
- [58] Offsetstone benchmark suite, 2003.
<http://www.address-code-optimization.org>.
- [59] D. Ottoni, G. Ottoni, G. Araujo, and R. Leupers, “Offset assignment using simultaneous variable coalescing,” In *ACM Transactions on Embedded Computing Systems (TECS)*, 5(4):864–883, 2006.
- [60] C. von Platen and J. Eker, “Feedback linking: optimizing object code layout for updates,” In *LCTES’ 06: Proceedings of conference on languages, compilers, and tools for embedded systems*, 41(7):2–11, 2006.
- [61] R. Panta, I. Khalil, and S. Bagchi, “Stream: Low overhead wireless reprogramming for sensor networks,” In *Proceedings of the 26th IEEE international conference on computer communications*, pages 928–936, 2007.
- [62] A. Perrig, R. Szewczyk, V. Wen, D. Culler, and J.D. Tygar, “SPINS: Security Protocols for Sensor Networks,” In *Proceedings of the ACM Annual International conference on mobile computing and networking*, 2001.
- [63] J. Polastre, R. Szewczyk, C. Sharp, and D. Culler, “The mote revolution: Low power wireless sensor network devices,” In *Hot Chips 16: A symposium on high performance chips*, 2004.
- [64] J. Polastre, R. Szewczyk, and D. Culler, “Telos: enabling ultra-low power wireless research,” In *Proceedings of the fourth international symposium on information processing in sensor networks*, 2005.
- [65] M. Poletto and V. Sarkar, “Linear scan register allocation,” In *ACM transactions on program languages and systems (TOPLAS)*, 21(5):895–913, 1999.
- [66] A. Rao and S. Pande, “Storage assignment optimizations to generate compact and efficient code on embedded DSPs,” In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation (PLDI)*, pages 128–138, 1999.
- [67] J. Regehr, A. Reid, and K. Webb, “Eliminating stack overflow by abstract interpretation,” In *ACM Transaction on Embedded Computing Systems*, 4(4):751–778, 2005. Also, Tinyos stack analysis, http://docs.tinyos.net/tinywiki/index.php/Stack_Analysis.
- [68] N. Reijers and K. Langendoen, “Efficient code distribution in wireless sensor networks,” In *Proceedings of the 2nd ACM international conference on wireless sensor networks and applications (WSNA)*, pages 60–67, 2003.
- [69] National Institute of Standards and Technology, “Announcing the Advanced encryption standard (AES),” <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>.

- [70] M. Ros and P. Sutton, “A hamming distance based vliw/epic code compression technique,” In *Proceedings of the 2004 international conference on compilers, architecture, and synthesis for embedded systems (CASES)*, pages 132–139, 2004.
- [71] M. Ros and P. Sutton, “A post-compilation register reassignment technique for improving hamming distance code compression,” In *Proceedings of the 2005 international conference on compilers, architectures and synthesis for embedded systems (CASES)*, pages 97–104, 2005.
- [72] R. Shah, “Vulnerability Assessment of Java Bytecode,” Thesis, Auburn University, 2005.
- [73] M. Sridharan, D. Gopan, L. Shan, and R. Bodik, “Demand-driven points-to analysis for Java,” In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA)*, 2005.
- [74] J. Steffan, L. Fiege, M. Cilia, and A. Buchmann, “Towards multi-purpose wireless sensor networks,” In *Proceedings of the 2005 systems communications*, pages 336–341, Aug. 2005.
- [75] A. Sudarsanam, S. Liao, and S. Devadas, “Analysis and evaluation of address arithmetic capabilities in custom dsp architectures,” In *Proceedings of the 34th annual conference on Design automation (DAC)*, pages 287–292, 1997.
- [76] Tinyos, <http://www.tinyos.net>.
- [77] B. Titzer, D. Lee, and J. Palsberg, “Aurora: scalable sensor network simulation with precise timing,” In *Proceedings of the 4th information processing in sensor networks (IPSN)*, pages 477–482, 2005.
- [78] O. Traub, G. Holloway, and M. D. Smith, “Quality and speed in linear-scan register allocation,” In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation (PLDI)*, pages 142–151, 1998.
- [79] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D.I. August, “Compiler optimization-space exploration,” In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 204–215, 2003.
- [80] L. Wang, “Mnp: multihop network reprogramming service for sensor networks,” In *Proceedings of the 2nd international conference on embedded networked sensor systems (SenSys)*, pages 285–286, 2004.
- [81] D. L. Whitfield and M. L. Soffa, “An approach to ordering optimizing transformations,” In *Proceedings of the second ACM SIGPLAN symposium on Principles & Practice of Parallel Programming (PPoPP)*, pages 137–146, 1990.
- [82] F. Ye, G. Zhong, S. Lu, and L. Zhang, “Gradient broadcast: A robust data delivery protocol for large scale sensor networks,” In *Wireless Network*, 11:285–298, 2005.
- [83] Y. Yu, L. J. Rittle, V. Bhandari, and J. B. LeBrun, “Supporting concurrent applications in wireless sensor networks,” In *Proceedings of the 4th international conference on embedded networked sensor systems (SenSys)*, pages 139–152, 2006.

- [84] M. Zhao, B.R. Childers, and M.L. Soffa, “Predicting the impact of optimizations for embedded systems,” In *Proceedings of ACM SIGPLAN Conference on Language, compiler, and tool for embedded systems (LCTES)*, 1C1, 2003.
- [85] Y. Zhang and R. Gupta, “Data compression transformations for dynamically allocated data structures,” In *Proceedings of the 11th International Conference on Compiler Construction (CC)*, pages 14–28, 2002.
- [86] X. Zhuang, C. Lau, and S. Pande, “Storage assignment optimizations through variable coalescence for embedded processors,” In *Proceedings of the 2003 ACM SIGPLAN conference on language, compiler, and tool support for embedded systems (LCTES)*, pages 220–231, 2003.
- [87] X. Zhuang and S. Pande, “An optimization framework for embedded processors with auto-addressing mode,” In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(11), 2010.