SOFTWARE UPDATE MANAGEMENT IN WIRELESS SENSOR NETWORKS

by

Weijia Li

B.S., Nanjing University, 1999

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

2010

UNIVERSITY OF PITTSBURGH

COMPUTER SCIENCE DEPARTMENT

This dissertation was presented

by

Weijia Li

It was defended on

October 7, 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, 2010

Wireless sensor networks (WSNs), composed of a large number of low-cost, battery-powered sensors, and a relatively powerful sink node, have recently emerged as promising computing platforms for many non-traditional applications. The code running on the sensors is preloaded to them before deployment, but it still needs updates for bug fix or feature enhancement. The code update patches are often transmitted via wireless channels, because the sensors are usually left unattended after deployment.

As the patch is transmitted via wireless communication, the energy consumed in the software update can be significant, especially when it happens frequently. Because the sensors are powered with batterers thus running with limited energy supplies, consuming too much energy in the software update procedure can shorten the life time of the sensors as well as the WSN. Therefore, designing an efficient software update method for WSNs is very critical.

This dissertation proposes an efficient WSN software update framework that contains novel solutions to this problem. A WSN software update procedure can be divided into two phases: the patch preparation and patch distribution. In the former phase, the new binary image is first generated by a compiler, and then the binary level *diff* between the old and new binary is used to form the update patch. In the latter phase, the patch packets are disseminated from source to destinations in the network. In the patch preparation phase, the proposed framework diminishes energy consumption of WSN software update by decreasing the patch size. It first uses an update-conscious compiler (UCC) that strives

to match the old compilation decisions when generating the new binary. This technology improves the binary level similarity, thus, reduces the *diff* that needs to be incorporated in the patch. It then uses a well designed patch formating method that can present multiple binary level differences in one primitive, which makes the patch even smaller. The experimental results show that these two methods workings together can reduce the patch size by 81% for general purpose applications and 61% for DSP applications. This dissertation proposes a multicast-based code redistribution protocol (MCP) to effectively disseminate the minimized patch to individual sensors. By saving the routing information on-site and using caching recently received packets, MCP can achieve a faster code delivery with less network traffic. The experimental results show that MCP reduces the network traffic by 56% and the code distribution time by 73%.

In summary, this dissertation solves an important problem in WSN study, and it is also the first research into update-conscious compilation techniques. The designed software update framework will benefit all the WSN users by making the software update procedure faster and more energy efficient. In the future work, more update-conscious compilation techniques will be approached. Also, the proposed techniques can be adapted to other hardware platforms, e.g., smartphones. Similar energy savings are expected to be seen, while applying them to other platforms.

TABLE OF CONTENTS

LIST OF FIGURES

1.0   INTRODUCTION

Wireless sensor networks (WSNs) have recently emerged as a promising computing platform for many nontraditional applications, such as monitoring wildfires, monitoring oceanic life, and battlefield surveillance. A WSN usually consists of hundreds of sensor nodes that are equipped with the sensing devices to measure the physical phenomena, such as temperature, humidity, pressure, or movement of objects. Sensing results are constructed into data packets and routed back to sink nodes, which are typically more powerful, user accessible and have fewer energy constraints.

(a) Mica2 sensor   (b) Mica2 block diagram
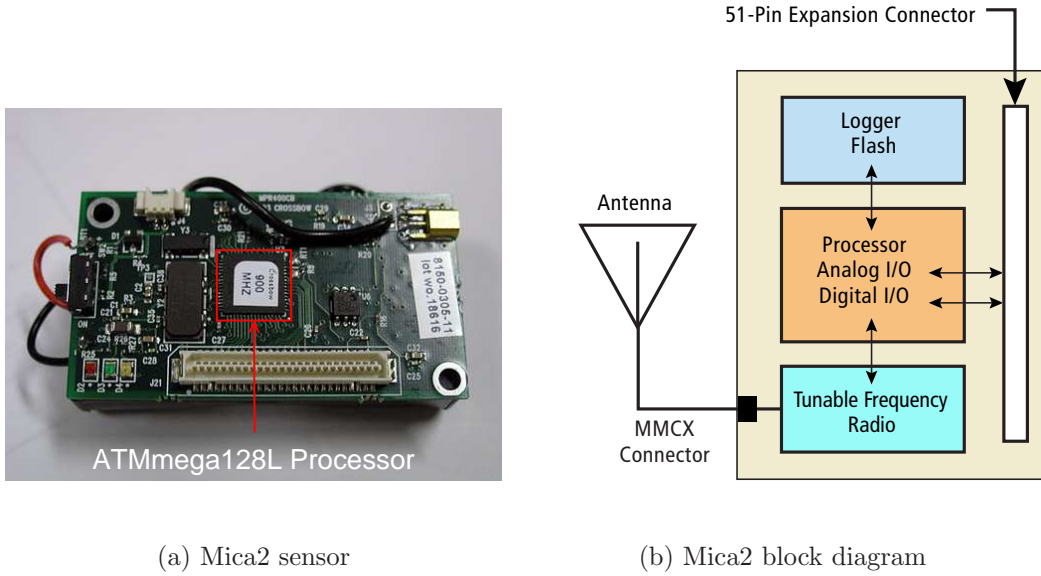
Figure 1: Mica2 sensor and the block diagram.

The sensor nodes are mainly equipped with processors, sensing devices and communication transceivers. Because they have limited power supply from their batteries, sensor

nodes [20, 21, 22, 23] are often equipped with a single low power consumption processor. For example, a Mica2 [20] mote shown in Figure 1, is equipped with an 8MHz ATmega128L processor to process the sensed data.

Due to technological advancement, sensors equipped with multiple chips [24] have been proposed recently. Shown in Figure 2, Imote2 [24] developed by Intel has a digital signal processor (DSP) chip on the mote, in addition to the CPU core, to support image and video operations. The multi-chip sensor is now able to pre-process the multimedia content sensed by the equipped camera and microphone, so that the package that needs to be sent back to the sink contains only a summary of the sensed results.
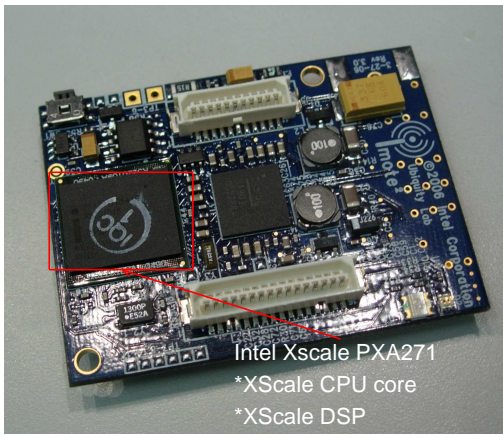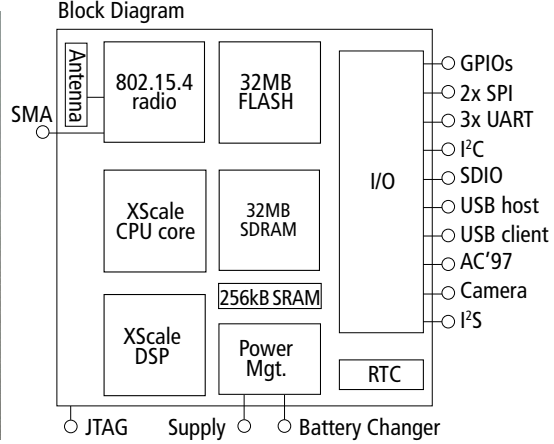
(a) Imote2 sensor   (b) Imote2 block diagram

Figure 2: Imote2 sensor and the block diagram.

The availability of multi-chip sensors poses various design challenges. 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 can support application concurrency in one network infrastructure [55, 61]. Furthermore, MA-WSNs are able to execute different applications interleavingly on one sensor node.

Compared to single-application wireless sensor networks (SA-WSNs), MA-WSNs have many advantages in efficiency and flexibility. For example, MA-WSN can be deployed in

a national park to monitor both wildfires and animal movement. A greater proportion of sensors can be set to monitor animal movement during seasonal migration, and more sensors could be set to monitor wildfires during the summer, when wildfires are more likely. By using the same network infrastructure for both applications, MA-WSNs achieves two goals. First, they lessen the investment needed to deploy multiple sensor networks in the same area; and second, they can adapt to the changing environment and adjust coverage according to need.

Software needs updates in both SA-WSNs and MA-WSNs for various reasons. There are two circumstances that can cause software update in WSNs, software upgrade and software switch.

Software upgrade. In both SA-WSNs and MA-WSNs, the applications running on the sensors may need to be upgraded after the deployment. Bug fixes and feature enhancements are the common reasons for software upgrades. Updates are particularly frequent when applications are still under development because testing and debugging may take several rounds until the code is stable. For example, the WSN may be deployed in an unfamiliar area so that preliminary data may help scientists better calibrate their sensing applications. I formulate software upgrade problem as the problem of updating code or data of a running application in a WSN. As shown in Figure 3, software upgrading involves binary code generation on the sink node, patch deployment and image replacement on the sensor nodes. Because the sensors are usually left unattended after deployment, the patch deployment can
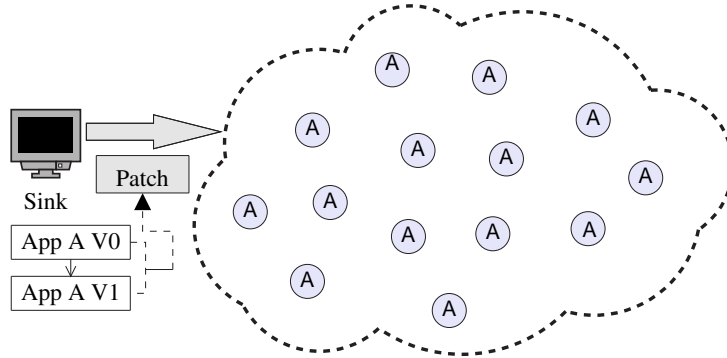
Figure 3: Software upgrade in a WSN.

only be done via wireless communication. This is an expensive operation in WSNs. For large WSNs, where the code source cannot reach destinations via broadcasting, the package has to

be transmitted hop-by-hop in the network, which consumes a significant amount of energy. A recent study [10] has shown that the energy used per bit sent over one hop in a WSN is about the same the energy used when executing 1000 instructions. As the sensor nodes are running with limited power supplies, it is essential to conserve the energy in a WSN during software updates, especially when such updates are frequent.

One possible solution to this problem is to reduce the number of bytes that need to be transmitted during software upgrades. Because the update patch is actually the binary level difference between the old and new code image, minimizing the binary level difference while generating the new binary will reduce the patch size, thus reducing the energy consumed during the transmission of the patch in the WSN. Patches can also be formatted to reduce their sizes. A good patch distribution protocol can also reduce the energy consumed in the software upgrade procedure.

Software switch. For SA-WSNs, software upgrade is the major reason to update the binary image on the sensors, but for WA-WSNs, there is another code update circumstance that can happen. In order to support application concurrency, multiple code images can be preloaded to the sensor nodes before deployment, and the sensors are able to switch between them upon request from the sink node. However, because of the memory size limitation, not all of the code images can be stored on the sensors. This will require the sensors to fetch the binary of the wanted yet unavailable application from somewhere else. The source can be the sink node, or the neighboring sensors that own this code image. Shown in Figure 4, while doing software switch, only a subset of the sensors in the network need to download the application. This is different from the case of software upgrades, where all the sensors need to download the new image. Also, both the sink node and the other local sensors may act as code sources, while in a software upgrade, only the sink node can be the source.

As in the software upgrade problem, in order to reduce the energy consumption during the software switch, an energy-efficient patch routing design and a good patch format design are both desired. As different sensor applications may share the same components, there are usually identical parts between the binaries of those applications. For example, in the sensor operating system TinyOS [5], there is a component called "Leds" that manages the led lights installed on the sensors, and this component is commonly used in many sensor applications
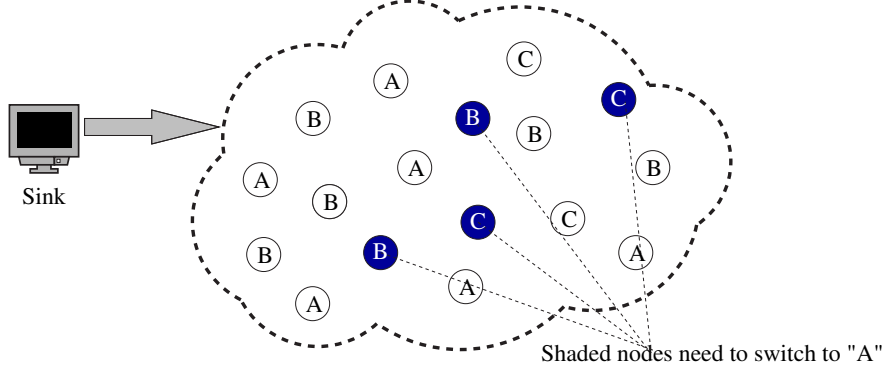
Figure 4: Software switch in a WSN.

such as "Blink" and "CntToLeds". One possible solution is to generate the same binary for the shared part, and use differential patching to transmit only the binary difference between the two applications rather than the entire new binary.

As discussed above, both software upgrades and software switches can happen frequently in WSNs. The patch transmission heavily relies on wireless communication, so it is a costly execution in WSNs. One of the great benefits that WSNs provide is spontaneity. Once the WSN is set up, it can sense and report the environmental information automatically. However, the high power consumption in the software update procedure will affect the lifetime of the WSN. As the sensors are usually hard to access after deployment, it may be difficult or even impossible to replace their batteries. New sensors may be purchased to replace the old ones, but this is neither economical nor environmentally friendly. Therefore, how to design an efficient software update framework is an important problem to study in WSN research.

1.1   OVERVIEW OF THIS RESEARCH

Regardless of whether the software update is caused by a software upgrade or a software switch, there are three 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

disseminated to the WSN. After each sensor receives the complete patch, it will rebuild the target binary, load it to program memory and start running it.

In order to solve the software update problem in WSNs under all the resource constraints, this dissertation presents a software update management framework, as shown in Figure 5. The sink node represents a computer server that is not resource-constrained. The sensor network consists of a large number of sensor nodes with resource limitations in energy, memory size, network bandwidth, time, and CPU.
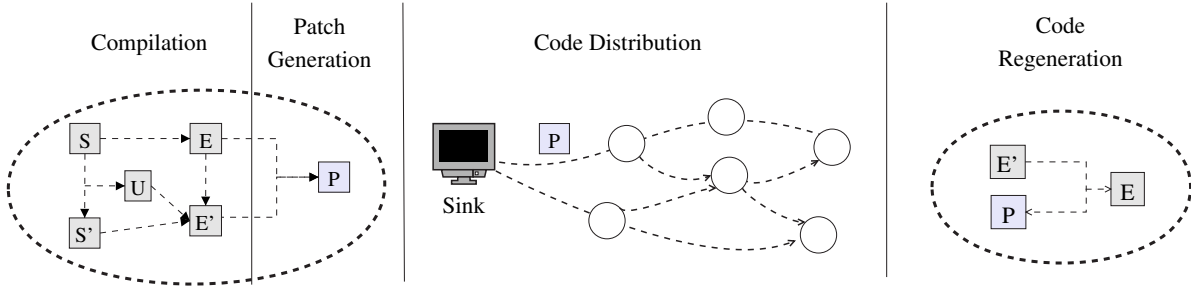
Figure 5: Software update framework overview.

The compiler first compiles the source code into an executable binary *E'*. Instead of sending the new binary image, a small patch *P'* that contains only the difference between *E'* and the original binary *E* is distributed over the network. After code distribution is complete, the target sensor regenerates the new binary code *E'* by combining the received patch *P* and the preloaded binary base *E*.

This framework includes three major components.

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

In the new version of the source code, the source level statements can be divided into two categories, changed statements and unchanged statements. The modified source statements will produce binary level differences, and these binary changes are difficult to avoid, yet the source level statements that are not changed may also create binary differences. This is

6

because in the code generation phase, the compiler may produce different binaries that have the same semantic meanings.

In this dissertation, I will propose update-conscious compiler (UCC) techniques that read the old source and binary to get the compilation choices of the old version, and use them as hints while generating the new binary. Shown in Figure 5, the 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'. A similar compilation option is chosen while generating the new binary; thus, this technique can improve the code similarity between the old and new binaries.

This dissertation focuses on the UCC register allocation and data allocation schemes for general purpose and DSP applications. It proposes ILP-based UCC register allocation and heuristic-based UCC data allocation for general purpose compilation. The incremental coalescing single/general offset assignment algorithms are designed for the DSP compilation as the UCC data allocation and UCC address register allocation schemes.

The goal of a UCC is to minimize the binary level differences, so it may not generate code that runs as efficiently as code generated by the conventional compilers. In fact, a UCC trades the run-time code performance for binary similarity; it saves energy during software updates but wastes energy during the run-time. If we consider the total energy consumed during both the software update and the execution before the next update, a UCC may not save the energy for the whole procedure and prolong the WSN's lifetime. In order to solve this problem, I studied the trade-off between the binary code differences and run-time performance. An adjustable UCC scheme is presented in order to achieve the a good trade-off and optimize for total energy use.

Patch generator. After the new binary E' is generated, it is then compared with the old binary E to generate the update patch. The binary level differences are described in a highly condensed script P. From the preliminary experimental results, I found that multiple binary level differences can result from the same cause, e.g., instruction insertion or removal can cause destination address shift for the branch instructions. Including the common cause instead of the individual changes should be able to reduce the patch size. However, if the update script design is too complicated, it will require more effort on the sensor side to

decode and regenerate the target new binary. This dissertation presents several sets of the script primitives, and discusses the trade-off between the patch transmission effort and the sensor side decoding effort.

Code distribution protocol. The code distribution protocol disseminates the patch packets to the destination sensors that need the software update. The code source can be either the sink node or the other sensors that own the requested binary image. This dissertation presents a code distribution scheme that works for both software upgrades and software switches.

The presented network protocol also runs under the WSN constraints. The wireless links in WSNs are not stable. Both the communications and nodes are unreliable because of the environment where the WSN is deployed and the limited energy resources of the sensors. Therefore, the network protocol has to be robust enough to tolerate link failure and node failure. Because of the high bandwidth and memory usage, the sensors may not be able to perform the sensing applications during the software update procedure. In order to reduce the down time of the sensors, the software update procedure is desired to be as fast as possible. Thus, the code distribution protocol design should disseminate the patch scripts through the WSN as quickly and with as little traffic as possible.

1.2   ASSUMPTIONS

The following assumptions are made to simplify the implementation of the software update framework in WSNs.

Multiple software update procedures on one sensor do not interleave with each other. Although it is possible that multiple applications can co-exist on one sensor node, there is a small chance that the software update to these applications happens at the same time. Also, because the sensors can only run one application at a time, it is not necessary to update them at the same time. Update sequence can be determined by the order of release time or the execution frequencies.

The binary is transmitted during software update. A typical compiler like GCC

can take over 100K memory space, so it is not practical to install compilers on the sensors. Therefore, we cannot send the source code to the sensors and let them compile the new binary. Some designs let the sensors run virtual machine instructions or high level instructions instead of the binary instructions in order to reduce the code size [26, 36, 41]. However, these methods introduce some run-time overhead, which may not be acceptable for a tightly resource-constrained embedded system, so optimizing the software update procedure for the virtual machine design is not discussed in this research. However, it is a future research opportunity.

The mapping between the source code and the binary can be built. Compiler optimizer may re-order the instructions in order to achieve better run-time performance. Therefore, the update-conscious compilation techniques should execute as a separated pass after the other compiler optimizations. We assume that the compiler can create the mappings between the optimized binary statements and the source statements to categorize the binary level differences into the functional changes and nonfunctional changes. The update-conscious compiler will then decrease the amount of the nonfunctional binary changes. Mappings between the unoptimized and optimized statements can be created by using the technology proposed by [33].

The number of executions of one application can be estimated. As the software engineers who design the application should be able to estimate how often one execution can be trigged, and the lifetime of one application can be estimated by the historical release logs, the number of executions that one application will do before retiring can be estimated. This number acts as a constant for each application in the energy consumption formula, and determines the compilation strategy that will be used to generate the binary in the adjustable UCC algorithms.

1.3   ORGANIZATION OF THIS DISSERTATION

The remainder of this dissertation is organized as follows. Chapter 2 presents background information on the general WSN software update and the three components of the proposed

framework, including traditional register allocation, data allocation design, WSN software update script primitive design and WSN code dissemination protocol design. Also, the relationship of this dissertation and prior work is discussed. Chapter 3 discusses the proposed UCC design, including the UCC register allocation and data allocation for general purpose applications and the UCC data allocation and address register allocation for DSP applications. Chapter 4 presents the script primitives that are used to summarize the binary level differences in the update patch. In Chapter 5, the patch distribution protocol is presented. Chapter 6 presents the experimental results and discusses the trade-offs. Chapter 7 addresses the directions for future research and the conclusion.

2.0   BACKGROUND AND RELATED WORK

In this section, I will first give a brief introduction to the related research about software update in WSNs, and then I will describe the background research that is related to the major components of my proposed framework.

2.1   SOFTWARE UPDATE IN WSNS

The existing WSN software update designs can be categorized according to *what* is to be transmitted over the network.

The naive solution is to transmit the complete new binary image to replace the old one on the sensors, e.g., Deluge (the default code distribution scheme in TinyOS [5]). The proposed schemes in this category mainly focus on how to package the new binary image and route the packets to the sensors under WSN constraints. However, since the new binary and old binary share some common code segments, transmitting the complete image is not necessary and is a waste of energy.

Another 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 more popular in WSNs [26, 34, 36, 45, 47, 51]. The major focus of this category is how to compare the two binary versions and generate a small edit script. However, because the code differences are derived from binaries generated

using the *conventional compiler's code generation methods*, with possibly some optimizations, a simple change in the source code may result in many changes in the final binary. This has limited the *diff*-based approaches to only small updates such as fixing a "bug" [51].

The third approach transmits the binary code at different levels. Some recent work introduced a small virtual machine [41] or a dynamic linker [26, 36] on the remote sensors. Instead of binary instructions, the code is represented at a higher level, e.g., virtual machine primitives, which can minimize the code difference in many cases. The tradeoff is that these approaches introduce high runtime overhead and may consume more energy in the long run.

2.2   COMPILER

Compiler is a program that can read a program in one language – the *source* language – and translate it into an equivalent program in another language – the *target* language. [7] The work flow of a compiler is shown as Figure 2.2.
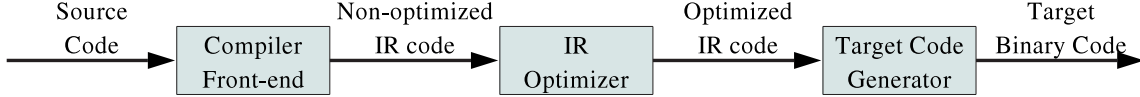
Figure 6: Compiler work flow

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.

My study [43] has shown a small local source code change might cause a big difference in the binary. This is because if the source code level difference is minor and local, when translating such source code into IR code, the difference will also be localized, assuming a one-one mapping can be created between the source code and IR code. However, transforming the IR code to the target binary code may propagate such local differences into global differences, by applying data allocation, code placement, register allocation, etc. Thus, my

update-conscious compiler (UCC) design will focus on the code generation phase. The goal is to let the compiler preserve the semantic meaning of the source program while generating a code image similar to a given one. Two key problems in code generation, register allocation and data allocation, are studied in this research.

2.2.1 Register allocation

Registers are the fastest computational unit on the target machine, however, usually there are fewer registers than the values that need to be held. The goal of register allocation is to determine *which value* should be held by *which register*. Because the values that are not held in registers reside in memory, and memory access takes longer time compared to register access, the efficiency of register allocation algorithm affects the execution efficiency of the program. Finding an optimal assignment of registers to variables is mathematically NP-complete. However, in the past twenty years, the register allocation problem has been extensively studied with 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 [13, 14, 16, 29]. To achieve fast compilation, linear-scan algorithms assign variables to available registers through a simple scan of the program, instead of constructing the interference graph [48, 58]. It was reported that linear-scan allocators generate similar performance level code as the graph coloring-based allocators, with a shorter compilation time required. Recently, the optimal or near optimal register allocation was formulated and solved through integer linear programming [8, 28, 30] or multi-commodity network flows [35].

However, all these register allocation algorithms listed above, target at generating code with better performance, in terms of less register spills, i.e., memory load and stores. In the WSN software update management concept, we want to design an UCC technique in order to reduce the size of the update between different versions of one program or two programs that share common components. Clearly, these traditional register allocation schemes do not fit in our requirement.

Update oriented register allocation. Bivens and Soffa proposed the incremental

register allocation (IRA) scheme [12] based on traditional graph coloring algorithm. While the software is update incrementally based on a previous version, the scheme only reallocates registers for the changed code, but preserves the assignment for unchanged code.

Because it is designed for incremental compilation, its goal is to save compilation time, when minor software update occurs. It may generate similar register allocation results as the previous version unintentionally, yet it always follows the original register allocation for the unchanged code, which may lose the code performance when the source code update is relatively large. Thus, even though such code similarity may cause energy saving in code distribution stage, more energy will be consumed in the future execution.

So besides considering the code similarity, the update-conscious register allocation (UCC-RA) scheme should also be adaptive to both small and large source code updates. The design goal is to achieve optimal overall energy consumption, which includes both code dissemination and future code execution.

Code compression oriented register allocation. Ros and Sutton proposed a post-compilation register reassignment technique [54]. It creates the mappings of the registers that are used in isomorphic instructions, and tries to replace one register with its mapping register. The design goal is to increase the code similarity between different components within one program, in order to improve Hamming distance based code compression [53]. The idea can be borrowed to design UCC-RA techniques. However, when the paired register is not available for the register replacement, the register replacement will be aborted.

Prior work and UCC-RA. This dissertation proposes the update-conscious register allocation (UCC-RA) scheme. Different from the traditional register allocation schemes, UCC-RA tries to match the register allocation result with that of the old version in order to reduce the *diff* between two versions. In order to achieve the overall energy savings including both the update energy and the execution energy, it does not necessarily follow the old register allocation result all the time. When the update level or the execution frequency is high, it will be more performance-conscious because the run-time energy consumption will dominate the overall energy consumption.

2.2.2 Data allocation

Data allocation assigns memory location to the variables in the program. General purpose compiler usually assigns memory slots to the variables according to the declaration order of the variables, because it does not affect the code performance or code size. However, for DSP applications, research has shown that the data allocation may also affect the code performance and code size [11, 44].

Addressing code generation in DSPs. Modern multi-chip wireless sensors have integrated DSP processors to support multimedia applications that process audio, video and communication signals. DSP processors strive to 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 incurs no extra cost. As a comparison, the *base-register-plus-offset* addressing requires two instruction words on 16-bit DSP processors e.g. AT&T DSP16xx [38].

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; otherwise an extra instruction is needed to update the address register.

Figure 2.2.2 shows an example of how the data allocation result affects the code generation. With out using the auto addressing instruction, we need two instructions (instruction 20 and 30) to load the value of A to $R1$ and then make the address register AR pointing to variable B. However, using the auto addressing instructions, the two operations can happen in parallel (instruction 20'), because the distance between the two consecutive memory accesses (to variable A and B) is 1. It saves one instruction in this example.
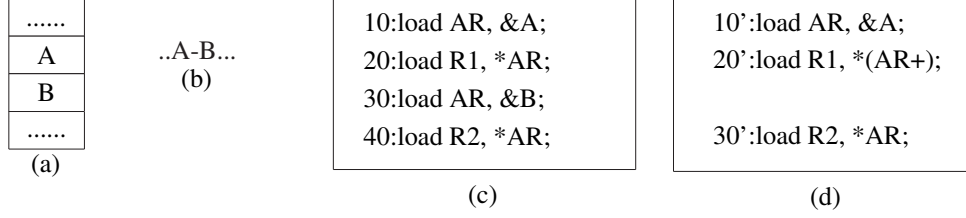
Figure 7: An example of DSP code generation. (a) Memory layout; (b) Access sequence of the variables; (c) Generated instructions without auto addressing; (d) Generated instructions with auto addressing.

Offset assignment. From the example, we can see that auto-addressing mode can increase the code performance and reduce the code size. On the other hand, the memory layout has to be optimal in order to achieve the best code performance and compression. The problem of assigning variables in memory was formulated by Bartley [11] and Liao *et al.* [44] as the simple offset assignment (SOA) problem, where there is only one AR, and general offset assignment (GOA) problem, where there are multiple ARs. A variety of heuristic algorithms have been proposed later [9, 15, 39, 40, 46, 49, 56, 62, 63].

The general solution of SOA is equivalent to finding the maximum weight Hamiltonian path[1] in the access graph [11, 44], where each vertex represents a variable; each edge between two vertices represents there is at least one consecutive access between the two related variables; and the weight of each edge shows the number of times that the two variables are consecutively accessed. GOA problem can be simplified as N SOA problems, where N is the AR number.

Offset Assignment with variable coalescing. Since many variables have short live ranges, variables that do not interfere with each other can be allocated in the same memory location, to furthermore reduce data memory size and improve code performance. An efficient offset assignment heuristic using variable coalescing is proposed by Ottoni *et al.* [46] and Zhuang *et al.* [62, 63]. In each iteration, either an unselected edge is selected to be on the

---

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

16

Hamiltonian path, or two vertices are chosen to be coalesced until no more vertices can be coalesced and the Hamiltonian path is built.

Prior work and UCC-DA. The design goal of the current offset assignment schemes is to generate efficient code with compact code size and/or 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 variable 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.

This dissertation proposes the update-conscious data allocation (UCC-DA) scheme to solve this problem. It uses the data allocation result from the old version as a hint while generating that of the new version, to reduce the *diff* between two versions. The code efficiency and code size are also considered in order to keep the code similarity without sacrificing too much performance.

2.3   PATCH GENERATOR

After the code compilation is finished, the sink node generates the patch code based on the binary code. There are several ways to prepare the patches.

Differential patching. When an update is an incremental improvement on a deployed function, such as bug fixing or parameter changing, the new image is often very similar as the old image. So instead of transmitting the complete image of the updated/new application, a differential patch between two images can be transmitted as the update. However, this method only works well for small updates. When the update is relatively large, the patch size may be large too.

Differential patching scheme can be used in our WSN software update framework design, because the UCC technique could reduce the binary level differences. As discussed before, a small register assignment or data assignment change could cause a large amount of binary

level differences which affect several instructions. The current differential patching schemes only present the instruction level differences in the script, which need to incorporate multiple instruction changes in the update script even though they are caused by the same reason. However, if the context-aware information, such as register assignment change or data assignment change is provided in the script, the sensors may be able to update the binary code by itself, which can significantly reduce the update patch size.

High level instruction. Patching code at higher semantic levels tends to generate smaller update script. Levis *et al.* showed that the code size is very short when they are represented using virtual machine instructions [41]. Marrón *et al.* proposed a scheme to produce separate object files for TinyOS [5] components and linked by sensors [45]. Dunkels *et al.* further proposed a dynamical linker for this systems [26]. Koshy *et al.* proposed to relocated modules and generate the binary using a remote linker [36]. A drawback of releasing code not in the binary format but the higher level language is that it requires extra runtime overhead, which might be not acceptable for tightly resource-constrained embedded system.

Compression. Compression algorithms, such as bzip2, compress, LZO, PPMd and zlib, can be used on the generated binary code to reduce the patch size. Simply using these existing compression algorithms on the binary code can reduce the patch size by 20%~70%. Further evaluations can be done to compare the overhead and performance of these compression algorithms running in the sensor environment. Selecting a high compression ratio algorithm that can work under the sensor constraints, will further diminish the patch size. However, this dissertation does not focus on the compression algorithm comparison and selection.

Prior work and this work. Because the sensors are running with limited energy supplies, I chose the least energy consuming patch generation scheme – differential patching as the base of my design. In addition to the simple primitives that are already proposed, I introduced the advanced and context-aware script primitives to present multiple binary level changes in one script statement. That will furthermore reduce the patch size.

2.4 DISTRIBUTION PROTOCOL

After the patch generation stage, the patches are ready to be distributed over the network. Many code distribution protocols [31, 32, 42, 60, 61] 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 efficient data flooding scheme in WSN called SPIN [31] uses a three-way handshaking protocol, shown in Figure 2.4. Each sensor broadcasts the software information (ADV messages) as advertisements. The sensors that need to update its software send a request (REQ) message to the code owners, and then the code owners respond with the data messages.
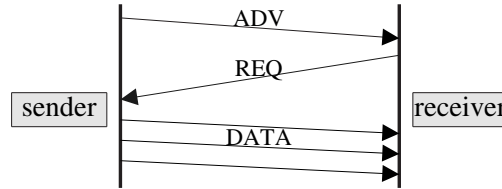
Figure 8: Basic code distribution protocol (SPIN).

Trickle [42] improves SPIN by adding periodical advertisement feature, which reduces the energy used in the advertise phase. Deluge [32] extends Trickle to support efficient flooding of large data especially code images, by dividing big code image into fixed sized segments, and the transmission of different segments can happen simultaneously in the network for rapid code prorogation purpose.

These protocols are only used to support the code update for SA-WSNs, so they assume that all the sensors will get the application updated eventually. Therefore, the advert messages only need to be broadcast one hop away. These protocols cannot handle the software update in MA-WSNs. Because MA-WSNs support concurrent application execution, only part of the sensors may be running the application that is recently updated. The distance between the requester and the source can be over one hop.

MA-WSN code distribution protocol. Melete [61] protocol is proposed to solve the

code distribution problem in MA-WSNs. It is a multicast-based protocol that only sends the code image to the sensors that request it instead of broadcasting it to all the sensors. Besides that, multi-hop code dissemination is also supported. The weakness of this scheme is that it is a stateless protocol, that does not store the routing to the source on site but relies on the REQ message to discover the routing, which may cause REQ message flooding in the network.

Prior work and MCP. This dissertation proposes a multicast-based code redistribution protocol (MCP). It solves the code distribution problem for both SA-WSNs and WA-WSNs. Different from Melete, each sensor stores the routing information on site, thus it can save the energy and time used to discover the route to the source. Because of the memory size limitation of the sensors, not all but the selected routes are stored on the sensors to balance the trade-off between performance and memory consumption.

3.0   UPDATE-CONSCIOUS COMPILER (UCC) TECHNIQUES

The conventional compilation takes the following steps to generate binary code from the source code, as depicted in Figure 9. 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.

- S—source code
- ir—intermediate representation
- IR—intermediate representation after optimization
- E—binary executable for S
- Symbols with ' are updated versions
- Δ is the difference between IR and IR'
- U is the update script
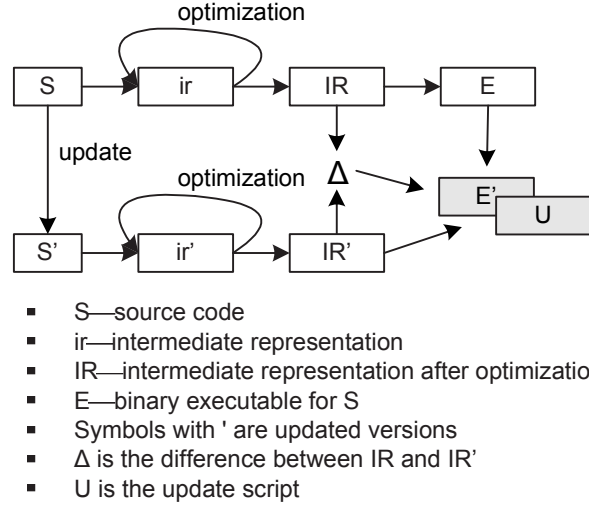
Figure 9: The sink-side update-conscious compilation.

The proposed UCC schemes are performed at the code generation stage, i.e. from IR to E. This helps to preserve the performance improvements from the optimization passes. Three update-conscious schemes are proposed for register allocation and data allocation phase in the code generation stage. For clarity, I assume that the optimization passes are

independent from these two phases, and other optimizations will be investigated in the 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 U, is small.

The binary difference U will then be transmitted over the network to the sensors. When the sensors receive the complete U, it will construct the target executable E' by combining U with old version executable E. This demonstrated in Figure 10.
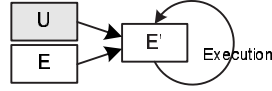
Figure 10: The sensor-side code update and execution.

3.1 UCC TECHNIQUES FOR GENERAL PURPOSE APPLICATIONS

In this section, I will discuss about the update-conscious compiler (UCC) techniques for general purpose applications. The UCC schemes for general purpose applications include the register allocation scheme, data allocation and the integrated scheme that combines both register allocation and data allocation. The goal is to generate the new binary image as similar as the old binary image as possible with the minimal run-time performance loss. I will discuss the detailed algorithms in this section.

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

The binary instructions can be updated due to data allocation result changes. For example, the load/store instructions that access a variable whose memory address is changed, need to be updated. Thus, one task of a UCC is to improve the data allocation similarity.

3.1.1.1 Data allocation problem for general purpose applications

Source:	Assembly:
uint_16 a;	; *a offset=0*
uint_16 b;	; *b offset=2*
uint_16 c;	; *c offset=4*
…	…
a=100;	li r1, 100
c = a + b;	ld r2, 0xa02
…	add r2, r2, r1
	st r2, 0xa04

(a)

Source:	Assembly:
~~uint_16 a;~~	; *b offset=0*
uint_16 b;	; *c offset=2*
uint_16 c;	; *d offset=4*
uint_16 d;	
…	…
a=100;	li r1, 100
c = 100 + b;	ld r2, 0xa00
d = b <<1;	add r2, r2, r1
	st r2, 0xa02
	lsl r2

(b)

Source:	Assembly:
uint_16 d;	; *d offset=0*
uint_16 b;	; *b offset=2*
uint_16 c;	; *c offset=4*
…	…
a=100;	li r1, 100
c = 100 + b;	ld r2, 0xa02
d = b <<1;	add r2, r2, r1
	st r2, 0xa04
	lsl r2

(c)

Figure 11: An incremental data allocation example. (a) Original source and assembly code; (b) New code and the update script; (c) Incrementally generated new code with a smaller update script.

The data allocation strategy can affect the similarity between different versions of binary, as illustrated in the example in Figure 11. In the original code (Figure 11(a)), three variables *a, b*, and *c* are allocated with offset 0, 2, and 4 respectively, to a base address. Assume the code is updated by replacing variable *a* with a constant, and introducing a new variable *d*. The existing compiler may generate the data allocation scheme as shown in Figure 11(b), in which all variables are assigned with new offsets, resulting in three updated instructions. However, an update-conscious algorithm should put the new variable *d* in *a*'s old location, as shown in Figure 11(c), resulting in only one updated instruction. On the other hand, if there was no *d* in the new code and if the word taken by *a* was not claimed, I would waste the word in RAM or more if the function is recursively invoked on the call stack. This will increase the memory usage on remote sensors.

The memory space here refers to the RAM space, which is used to store the call stack. A typical wireless sensor has a 4KB RAM (Mica2 or MicaZ), used to store not only the call stack but also the data segment and the BSS segment. Figure 12 shows the sensor memory model. The size of the data segment and BSS segment can be calculated by using static analysis, but the stack size changes as the program executes. In order to make sure that

the stack will not overflow, the worst case stack size counting the memory waste cause by UCC-DA should satisfy the following equation.

$$stack\ region\ size \leq RAM\ size - (data\ segment\ size + BSS\ segment\ size) \qquad (3.1)$$
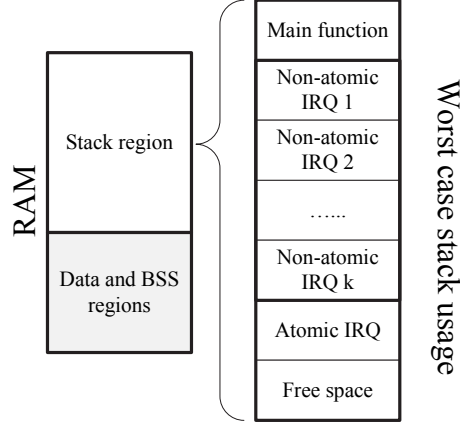
Figure 12: The sensor memory model.

3.1.1.2 UCC data allocation for general purpose applications

To address the problem of how to keep the data allocation similarity as well as the worst case call stack size to be lower than the available RAM size, I propose a *threshold-based data allocation* mechanism [43]. The intuition is to reuse the space of the deleted variables as much as possible, and when there are more new variables than the deleted variables we will create memory holes in the functions that have the least affect to the overall memory usage.

- 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 space will be left. There are two options to save this space:
  - relocate some old variables;
  - do not relocate.

  The first option does not waste the space resource on sensor node, but it needs to change the program code because of the variable reallocation. The second option incurs less code

changes but leaves "holes" in the call stack at runtime. As a hybrid of these two options, the proposed algorithm keeps the maximal data allocation similarity, under the constraint that the total wasted RAM space is less than a given threshold — $SpaceT$. This threshold can be calculated by computing the size of each segment in RAM using traditional compilation method and subtract that from the RAM size. For ease of illustration, the proposed algorithm elaborates on the procedures for variables of word type only. The principle can be applied to other data types such as array and composite structures similarly.

The detailed algorithm is shown in Algorithm 1.

Algorithm 1 UCC-DA for general purpose applications.

Input: Function 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:     $NumOfDelV_i \leftarrow$ the total number of deleted variables in P_i;
4:     $NumOfNewV_i \leftarrow$ the total number of new variables in P_i;
5:     $NumOfInsts_i \leftarrow$ the projected maximal simultaneous instances of P_i;
6:     **if** $NumOfDelV_i \leq NumOfNewV_i$ **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:        $ExtraSpaceSize_i \leftarrow NumOfDelV_i - NumOfNewV_i$;
12:        $TotalWastedSpaceSize \mathrel{+}= ExtraSpaceSize_i \times NumOfInsts_i$;
13:     **end if**
14: **end for**
15: **while** $TotalWastedSpaceSize > SpaceT$ **do**
16:     $Max\_Factor \leftarrow 0$;
17:     **for** $P_i \in P[]$ AND $ExtraSpaceSize_i > 0$ **do**
18:        $Usage_i(last) \leftarrow$ the usage of the last variable in P_i;
19:        $Factor_i \leftarrow \frac{NumOfInsts_i}{Usage_i(last)}$;
20:        **if** $Factor_i > Max\_Factor$ **then**
21:           $Max\_Factor \leftarrow Factor_i$;
22:           $To\_Move \leftarrow i$;
23:        **end if**
24:     **end for**
25:     Move the last variable in function To_Move to fill up a memory "hole";
26:     $TotalWastedSpaceSize \mathrel{-}= 1 \times NumOfInsts_i$;
27: **end while**

First, it collects the following profiles for each function $P_i (i \geq 0)$ in the program.

$NumOfDelV_i$	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;
$Usage_i(a)$	the usage of variable 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 allocate 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 formular:

$$ExtraSpaceSize_i = NumOfDelV_i - NumOfNewV_i \tag{3.2}$$

and moves to the next step.

In this step, it adjusts the data allocation by incrementally relocating the *last* variable in each function. It keeps moving the last variable into a "hole" left by variable deletion, until all the "holes" are filled. That is,

$$\sum_{\forall Pi} ExtraSpaceSize_i \times NumOfInsts_i \leq SpaceT \tag{3.3}$$

While deciding which function needs to move the last variable up to fill up the "hole", functions are ordered by two factors. One is the number of usages of the last variable $Usage_i(last)$ and another one is the number of instances that the function can have on stack $NumOfInsts_i$.

If the last variable in the function is used more often, such data allocation move will cause more instruction updates, so we want to move the last variable in the function whose last variable is rarely used. Such that, less code update will be triggered.

Another factor is the memory waste. As shown in Figure 12, if a function has more than one instance on stack. For example, it can be called by both the main function and the interrupt handler(s). One word memory waste in this function may cause more than one word RAM waste. Thus, when we have to leave a hole in the memory to keep data allocation

26

similarity, we want to pick up the function that has the smallest number of instances on the stack, because such decision will cause the least amount of RAM space waste.
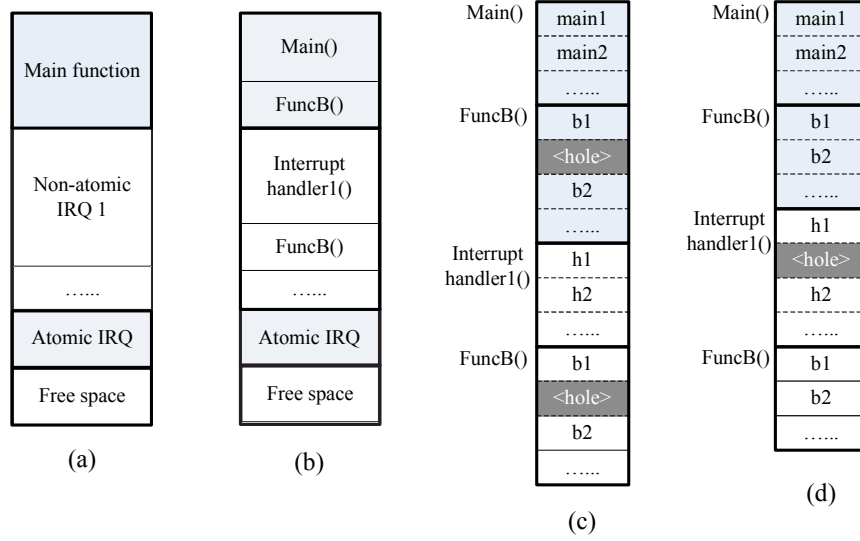
Figure 13: An example of data allocation for general purpose applications.

Figure 13 shows a memory usage example. $FuncB()$ is called by both $Main()$ and $Interrupt\_handler1()$, so it has two instances on the stack, while the other functions only have one instance. Wasting one word in $FuncB()$ will cause two word waste in RAM, while wasting one word in $Interrupt\_handler1()$ will cause only one word waste in RAM. Thus, in order to save RAM usage, we should move the last variable of function $FuncB()$ instead of $Interrupt\_handler1()$.

With such consideration, the function j that we pick should satisfy the following formular.

$$\frac{NumOfInsts_j}{Usage_j(last)} = MAX(\frac{NumOfInsts_j}{Usage_i(last)}) \quad (\forall j, ExtraSpaceSize_j > 0) \tag{3.4}$$

After we decide which function we will pick, we then relocate the last variable in procedure j to one deleted memory word. By doing so, we can shrink the maximal runtime memory usage by $NumOfInsts_j$ (as it is the last variable in that procedure), and incur less code changes (as the variable with less usage is selected). We then decrement $ExtraSpaceSize_j$ and continue this step until equation (3.3) is satisfied.

27

For example in Figure 11, if d is not introduced, we will reuse a's space with c if $SpaceT = 0$, i.e. no wasted space. This will result in two updated instructions related to c and d respectively. This code still outperforms the default scheme in Figure 11(b) which requires three instruction updates.

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

Besides the data allocation results, the register allocation results can also affect the similarity between generated binary images. Instructions need to be updated if the assigned registers are changed. Thus, another task of a UCC is to produce similar register assignment when generating the new binary.

3.1.2.1 Register allocation problem for general purpose applications

Figure 14 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 14(a)). Assume a small code change extends b's live range into a's. If there are enough free registers, a modern register allocator will assign different registers to them, as depicted in Figure 14(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 14(c). By comparing these two solutions, it is clear that while the solution (b) achieves better code quality, the solution (c) results in less update cost. The discrepancy in energy consumption between data transmission and instruction execution makes the solution (c) more appealing as it consumes less energy unless the code is very frequently executed, or the update is extremely rare.

```
(1) a = ...                          (1) a = ...                          (1) a = ...
...                                  ...                                  ...
(5) ... = a + ...  ⎤ a: R1          [(5) b = a + ...] ⎤ a: R1          [(5) b = a + ...] ⎤ a: R1
...                ⎥                 ...               ⎥                 ...               ⎥
(10) ... = a       ⎦                (10) ... = a       ⎦                (10) ... = a       ⎦
                                                                        ...                  ⎤ b: R2
...                                 ...                                 [ mov R2, R1]
(12) b = ...       ⎤               (12) b = ...       ⎤ b: R2          (12) b = ...       ⎤ b: R1
...                ⎥ b: R1          ...                ⎥                ...                ⎥
(15) ... = b       ⎦               (15) ... = b       ⎦               (15) ... = b       ⎦

(a) variables a and b               (b) a and b's live ranges           (c) split b's live range and
are assigned to use R1              overlap after the update            assign a new register
```
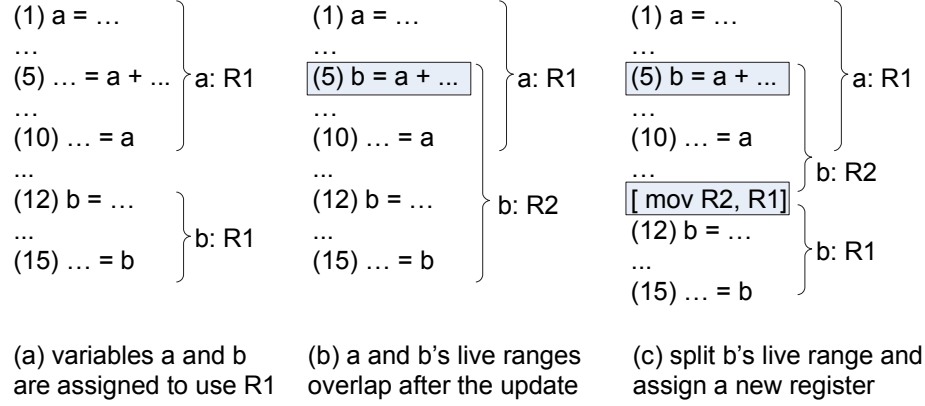
Figure 14: An example of register allocation for general purpose applications. (a). Variable a and b are assigned with register R1; (b). The live ranges of a and b overlap with each other after the code update; (c). Split b's live range and assign a new register to it.

3.1.2.2 UCC register allocation for general purpose applications

The basic idea of UCC register allocation (UCC-RA) [43] is to retain mostly the old register assignments and perform new register allocations to the changed and new instructions *with preferences to the decisions for the given 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 the the semantic correctness.

While doing UCC-RA, each variable in the input chunk is tagged with the register name that was assigned 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 [30], the UCC-RA problem can be formulated as a non-linear integer programming problem. The general idea of how to select the decision variables, formulate the constraints and the objective function is addressed the below.

The decision variables. I use a set of decision variables that represent the register assignments we need to make at each program point. The decision variable is defined as 3.5.

$$X_{op.v.s}^{Ri} = \left\{ \begin{array}{lll} 0 & : & \text{Assertion is true.} \\ 1 & : & \text{Assertion is false.} \end{array} \right. \tag{3.5}$$

$$\begin{array}{c|l} op & \text{operation;} \\ v & \text{variable;} \\ s & \text{statement;} \end{array}$$

The decision variables $X_{op.v.s}^{Ri}$ can be 0 or 1. With the value assigned to be 1, it means that register Ri is assigned to variable v at statement s for operation op, and 0 otherwise. For example, when decision variable $X_{def.v.s}^{Ri}$ is set to 1, it means that the variable v is defined at statement s, and register Ri is assigned to hold the value of variable v. In the example

shown in Figure 14(b) statement (1), the decision variable that is used to determine whether to allocate variable a in $R1$ can be written as $X^{R1}_{def.a.1}$. **The objective function.** Let us use a simple example in Figure 14 to explain the proposed procedure. The code contains several instructions: the first two are the definitions of variable a and b respectively, while the third one uses both variables. Let us assume at statement (6), a is dead but b is still alive, and the preferred-registers of a and b are $R1$ and $R2$ respectively.

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

Figure 15: The decision variables used in UCC-RA.

For the code chunk in Figure 14(a), we first introduce a set of decision variables that represent the register assignments we need to make at each program point. For example, If variable a is allocated in register $R1$ at statement (1), then we have $X^{R1}_{def.a.1} = 1$ and $\forall Ri, Ri \neq R1, X^{Ri}_{def.a.1} = 0$. Here $X^{Ri}_{def.a.s}$ is a decision variable to show if variable a is assigned to the register Ri at statement s. As another example, if we decided to insert an

instruction "*mov R2 to R3*" for b before statement (4), we set $X^{R2}_{mov.out.b.4} = 1$, $X^{R3}_{mov.in.b.4} = 1$, and all other *mov* decision variables $X^{*}_{mov.*.b.4}$ as 0. As discussed, such a *mov* instruction may be inserted to release $R2$ for other variables, or to match the old assignment of b to $R3$ after statement (4). Figure 15 shows a full list of decision variables that we used in UCC-RA.

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

The constraints. With the defined decision variables, we 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.

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

$$\sum_{\forall Ri} X^{Ri}_{def.a.s} = 1. \tag{3.6}$$

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

$$\sum_{\forall Ri} X^{Ri}_{mov.out.a.s} \leq 1$$
$$\sum_{\forall Ri} X^{Ri}_{mov.out.a.s} = \sum_{\forall Ri} X^{Ri}_{mov.in.a.s} \tag{3.7}$$

At a statement s, variable a may be loaded from the memory, or come from inter-register movement. After defining the variable, the value in the register may be spilled to the memory, or moved to another register, or stay for later use. Thus we have:

$$X_{st.a.s}^{Ri} \leq X_{def.a.s}^{Ri} + X_{mov.in.a.s}^{Ri}$$
$$X_{mov.out.a.s}^{Ri} \leq X_{def.a.s}^{Ri}$$
$$X_{cont.a.s}^{Ri} \leq X_{def.a.s}^{Ri} + X_{mov.in.a.s}^{Ri} \tag{3.8}$$

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

$$X_{cont.a.s}^{mem} \leq \sum_{\forall Ri} X_{st.a.s}^{Ri} \tag{3.9}$$

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

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

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 it is the last use, we have one of the following two constraints:

$$X_{use.a.s}^{Ri} + X_{useCont.a.s}^{Ri} \leq X_{cont.a.(s-1)}^{Ri} + X_{ld.a.s}^{Ri} + X_{mov.in.a.s}^{Ri}$$

$$X_{last.a.s}^{Ri} \leq X_{cont.a.(s-1)}^{Ri} + X_{ld.a.s}^{Ri} + X_{mov.in.a.s}^{Ri} \tag{3.11}$$

Since we only generate load spill, or inter-register movement before the use point, we have:

$$\sum_{\forall Ri} X_{ld.a.s}^{Ri} \le X_{cont.a.(\mathbf{s-1})}^{mem}$$

$$\sum_{\forall Ri} X_{mov.out.a.s}^{Ri} \le X_{cont.a.(\mathbf{s-1})}^{Ri} \tag{3.12}$$

The following constraints are used to ensure that one register is assigned to only one variable at a time.

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

$$U_{a.s} = \begin{cases} X_{def.a.s}^{Ri} & : & \text{if } a \text{ is defined at } s \\ X_{use.a.s}^{Ri} + X_{useCont.a.s}^{Ri} & : & \text{if } a \text{ is used at } s \\ X_{lastUse.a.s}^{Ri} & : & \text{if } a \text{ is used at } s \text{ and no longer used in the later instructions} \end{cases} \tag{3.13}$$

$V_{a.last\_s}$ is defined to show the register assignment information of the last access point of variable a. Depending on whether instruction $last\_s$ is a defintion point or use point of variable a, $V_{a.last\_s}$ is calculated in two different ways.

$$V_{a.last\_s} = \begin{cases} X_{cont.a.last\_s}^{Ri} & : & \text{if } a \text{ is defined at } last\_s \\ X_{useCont.a.last\_s}^{Ri} & : & \text{if } a \text{ is used at } last\_s \end{cases} \tag{3.14}$$

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

$$\sum_{\forall var} V_{var.last\_s} + U_{a.s} \le 1 \tag{3.15}$$

For example, the following equations show the constraints at statement (2) in Figure 14:

$$U_{b.2} = X_{def.b.2}^{Ri}$$
$$V_{a.last\_use} = X_{cont.a.1}^{Ri}$$
$$thus, \quad X_{def.b.2}^{Ri} + X_{cont.a.1}^{Ri} \le 1 \tag{3.16}$$

34

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

$$\sum_{\forall var} U_{var.s} \leq 1 \tag{3.17}$$

For example, the following constraints at statement (6) in Figure 14:

$$U_{a.6} = X^{Ri}_{lastUse.a.6}$$
$$U_{b.6} = X^{Ri}_{use.b.6} + X^{Ri}_{useCont.b.6}$$
$$thus, \quad X^{Ri}_{lastUse.a.6} + X^{Ri}_{use.b.6} + X^{Ri}_{useCont.b.6} \leq 1 \tag{3.18}$$

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

$$X^{Ri}_{use.(a).s} = X^{R_{i+1}}_{use.(a+1).s}$$
$$X^{Ri}_{use.(a+1).s} = X^{R_{i+1}}_{use.(a+2).s}$$
$$X^{Ri}_{use.(a+2).s} = X^{R_{i+1}}_{use.(a+3).s} \tag{3.19}$$

At the boundary of changed and unchanged code chunks, and at the merge point of control flows, we insert inter-register move instructions to make sure that the values are in proper registers before their next uses. In our future work, instead of performing inter-register movements, we will introduce constraints similar to those in [30] for the merge point of control flows.

The objective function. The goal of our integer programming is to minimize the objective function on total energy consumption, as expressed in equation (3.20) in Figure 16. The equation defines the total energy consumption of the changed IR chunk under different register allocation decisions. The notations used in equation (3.20) are listed below. Other terms are explained as follows.

$$E_{total} = chg(s) \times E_{changed\_IR} + (1 - chg(s)) \times E_{unchanged\_IR} + E_{spill} + E_{extra} \qquad (3.20)$$

where
$$E_{changed\_IR} = \sum_{\forall s}(freq(s) \times E_{exe}) + \sum_{\forall s}(E_{trans}) \qquad (3.21)$$

$$E_{unchanged\_IR} = \sum_{\forall s}(freq(s) \times E_{exe}) +$$
$$\sum_{\forall s}(1 - \prod_{\forall a} X_{def/use.a.s}^{prefer(a,s)}) \times E_{trans} \qquad (3.22)$$

$$E_{spill} = \sum_{\forall s,a,Ri}(freq(s) \times (X_{st.a.s}^{Ri} + X_{ld.a.s}^{Ri}) \times E_{exe}) +$$
$$\sum_{\forall s,a,Ri}((1 - spill(a, Ri, s)) \times (X_{ld.a.s}^{Ri} + X_{st.a.s}^{Ri}) \times E_{trans}) \qquad (3.23)$$

$$E_{extra} = \sum_{\forall s,a,Ri}(freq(s) \times X_{mov.in.a.s}^{Ri} \times E_{exe}) +$$
$$\sum_{\forall a,s,Ri}(X_{mov.in.a.s}^{Ri} \times E_{trans}) \qquad (3.24)$$

Figure 16: The objective function used in UCC-RA.

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 17: The notations used in the UCC-RA objective function.

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

for R1, 1-spill(a,R1,s)=0, $X_{ld.a.s}^{R1} + X_{st.a.s}^{R1}$=1

for Ri(Ri≠R1), 1-spill(a,Ri,s)=1,$X_{ld.a.s}^{Ri} + X_{st.a.s}^{Ri}$=0

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

$E_{unchanged\_IR}$ specifies the energy consumption due to unchanged IR instructions. Assume we have an unchanged IR instruction "a=a+b" and a and b's preferred-registers are $R1$ and $R2$ respectively. If the new allocation decision follows the old allocation scheme, then there is no dissemination cost, i.e. the same binary instruction "$add\ R1,\ R2$" is generated. If a is assigned to a different register, say $R3$, and we generate "$add\ R3,\ R2$", then this new instruction needs to be disseminated to replace the old one on the sensor. As shown in equation (12), this component is 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. Our UCC-RA targets at achieving overall energy efficiency, i.e. E_{extra} is positive only when we 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 $chg(s), freq(s)$, etc. are known for a given code chunk. Since equation (12) is non-linear, the above formulation of UCC-RA results in a mixed integer non-linear programming problem (MINLP) [18]. While the speed of MINLP solvers has been improved greatly in recent years [18], it is still much slower than solving a linear problem.

Our experiments results show that MINLP can be orders of magnitude slower than a linear problem of similar sizes, i.e., similar number of decision variables and constraint. We next discuss how to convert the MINLP problem to an ILP problem through approximation.

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

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

$$\sum_{\forall s}(1 - X_{use.a...}^{R1} + 1 - X_{use.b...}^{R2}) \times E_{trans} \times \delta \tag{3.25}$$

where $\delta = 3/4$, a coefficient that approximates the update cost. It is decided as follows. Assume each variable has equal opportunity of being assigned and not assigned to its preferred register. For the instruction with two variables a and b and preferred registers *R1* and *R2* respectively, there are four possibilities altogether:

- 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 case (i) 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 decides δ to be $3/4$.

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

3.1.3 Integration of UCC-DA and UCC-RA

When constructing the objective function of the UCC register allocation scheme (UCC-RA), the changed and unchanged IR statements are treated differently. For the changed IR statements, the code update cannot be avoided, thus, using UCC-RA scheme cannot gain any benefit for this type of statements. On the other hand, for the unchanged IR statements, whether to keep the register assignment the same as the old version or not will determine whether it is necessary to update the generated binary instruction(s). Notation $chg(s)$ is used to differentiate these two kinds of IR statements.

With the consideration of the update-conscious data allocation scheme (UCC-DA), we can see that for the unchanged IR statements, not only the register allocation results but also the data allocation results can affect the transmission energy consumption. If the unchanged IR statement is a memory access instruction, and the memory address of the operand is changed, no matter how the register allocation is done, the update energy cannot be waived. Thus, the objective function E_{total} for this statement should be formated using formular 3.21 instead of formula 3.22 in Figure 16.

Based on this observation, I propose two solutions to integrate UCC-DA and UCC-RA schemes for general purpose applications below. One is a near-optimal ILP based solution and another one is a light weight heuristic based solution.

3.1.3.1 ILP based integration

In order to integrate the UCC-DA and UCC-RA solutions for general purpose applications, let us first introduce a new decision variable to describe whether to reallocate a variable or not, then the related constraints and the revised objective function.

Decision variable $X_a^{realloc}$. I define a new decision variable that determines whether to allocate the variable in a different memory slot from the old version. The decision variable is written as $X_a^{realloc}$, and a presents the variable name. If the value is set to "0", the data allocation of this variable keeps the same; otherwise, it is changed.

Data allocation related constraints. As described in the UCC-DA algorithm 1, there are two constraints for UCC-DA decision variables, and I formulate those constraints using

the notations shown in Figure 18 as below.

$X_a^{realloc}$	whether to allocate a in a different memory slot from the old version;
$NumOfInsts_i$	the projected maximal simultaneous instances of procedure Pi;
$ExtraSpaceSize_i$	the wasted memory space of procedure Pi;
$NumOfDelV_i$	the total number of deleted variables in Pi;
$NumOfNewV_i$	the total number of new variables in Pi.

Figure 18: The notations used in the ILP based UCC integration.

The total wasted space on the call stack cannot go beyond the threshold $SpaceT$ 3.3. This constraint is formulated as equation 3.27. $NumOfDelV_i$ and $NumOfNewV_i$ are the number of variables that are removed and inserted in procedure Pi, so they are constants for each Pi. $NumOfInsts_i$ represents the projected maximal simultaneous instances of procedure Pi, which is also a constant. The new variables are firstly used to fill up the holes created by deleting variables. If there are fewer new variables than deleted variables, the unchanged variables in this procedure can be reallocated to fill up the memory "holes". The wasted memory space of procedure Pi is presented in equation 3.26, which is equal to the number of deleted variables minus the number of new variables, then minus the number of reallocated variables.

$$Extra_i = \begin{cases} 0 & : & DelV_i \leq NewV_i \\ DelV_i - NewV_i - \sum_{\forall a} X_a^{realloc} & : & DelV_i > NewV_i \end{cases} \quad (3.26)$$

$$\sum_{\forall Pi} Extra_i \times InstsNum_i \leq SpaceT \quad (3.27)$$

Another constraint is that it always allocates the last variable in each procedure first until the space constraint is met. This constraint is formulated as equation 3.28.

Integrated objective function. The objective function of the UCC-RA problem is formulated as equation 3.20. There are four parts in the objective function. The energy consumption of the ALU instructions is formulated as $E_{changed\_IR}$ and $E_{unchanged\_IR}$. The

$$\forall a, b \ \ old\_addr(a) \leq old\_addr(b) \Rightarrow X_a^{realloc} \leq X_b^{realloc} \qquad (3.28)$$

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 reformulated.

In the old formular, there were two factors that determine whether the corresponding load/store instruction needs transmission energy to update the instruction: whether there was a spill here in the old binary $(spill(a, Ri, s))$, 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 newly inserted instruction. Otherwise, the energy consumpiton depends on whether the register allocation results are the same as the old ones. In short, only when either variables $X_{ld.a.s}^{Ri}$ or $X_{st.a.s}^{Ri}$ as well as $spill(a, Ri, s)$ are set to be "1", the transmission energy can be saved.

Now, besides these factors, whether to reallocate the variable in memory will also affect the transmission energy. When the variable is reallocated in memory, which means $X_a^{realloc}$ is set to "1", the corresponsing load/store instruction needs update no matter how the other factors are set. In other words, the instruction update is not necessary, only when the following three conditions are all satisified:

- $spill(a, Ri, s)$ is set to be "1" which means this instruction was a *spill* instruction in the old binary;

- $X_{ld.a.s}^{Ri}$ or $X_{ld.a.s}^{Ri}$ are set to "1" which means that the new register allocation result is the same as the old binary; and

- $X_a^{realloc}$ is set to "0" which means the variable is not reallocated in memory.

Otherwise, this instruction needs to be transmitted. Thus, the energy consumption of the load/store instructions is then reformulated as equation 3.29.

As you can see, the energy function E_{spill}(3.29) is no longer a linear function due to

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

$$\sum_{\forall s,a,Ri} (1 - spill(a, Ri, s) \times (X_{ld.a.s}^{Ri} + X_{st.a.s}^{Ri}) \times (1 - X_a^{realloc})) \times E_{trans} \quad (3.29)$$

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

the addition of another decision variable $X_a^{realloc}$. In order to reduce the time spent solving the problem, we need to convert it into a linear function that will give us an approximated result but consumes much less time. Similar as what we did before in subsubsection 3.1.2.2, E_{spill}(3.29) can be rewritten as equation 3.30.

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

$$\sum_{\forall s,a,Ri} (1 - spill(a, Ri, s) \times (X_{ld.a.s}^{Ri} + X_{st.a.s}^{Ri} + 1 - X_a^{realloc}) \times \delta) \times E_{trans} \quad (3.30)$$

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

Here, δ is an coefficient that approximates the update cost when we convert the integer non-linear problem into an integer linear problem. Same as in subsubsection 3.1.2.2, we set δ here to be 3/4.

3.1.3.2 Heuristic based integration

The introduction of the memory reallocation decision variable $X_a^{realloc}$ will add N decision variables to the ILP problem, where N is the number of variables used in the program. This will increase the complexity of the ILP problem, thus, increase the compilation time. Based on this observation, I design another heuristic based integration algorithm, that does the

data allocation and register allocation separately. This scheme will not affect the complexity of the ILP problem, because no more decision variables will be introduced. The detailed algorithm is described as below.

Instead of solving both data allocation and register allocation problems simultaneously, the compiler will solve them one after another in sequential order. It does UCC-DA first and find out the variables whose memory addresses are to be changed. This information is then passed to UCC-RA. The memory access statements that access these variables will be considered as changed IRs instead of unchanged IRs.

$datachg(s)$ | if statement s is a memory access statement and the memory address
 | of the operand is changed from the old version.

Figure 21: The notation used in the heuristic based UCC integration.

I introduce another notation $datachg(s)$ shown in Figure 21 to describe whether the unchanged IR statement needs update because of data allocation result changes. After the UCC-DA is done, the compiler marks the variables whose memory location are changed. Then, the $datachg(s)$ parameters of those memory access statements that accesses any of these variables will be marked as "1". Otherwise, they will be marked as "0".

The objective function needs to be changed as shown in Figure 22. When both $chg(s)$ and $datachg(s)$ are set to "0" , the IR statement is treated as an unchanged statement.

$$E_{total} = chgIR(s) \times E_{changed\_IR} + (1 - chgIR(s)) \times E_{unchanged\_IR} + \qquad (3.31)$$
$$E_{spill} + E_{extra} \qquad (3.32)$$
$$chgIR(s) = 1 - (1 - chg(s)) \times (1 - datachg(s)) \qquad (3.33)$$

Figure 22: The objective function used in the heuristic based integration.

The heuristic based solution is easy to implement, however, the result may not be optimal. While doing UCC-DA, the compiler does not have any information of the UCC-RA result.

It assumes that reallocating a variable in memory will always increase code update overhead and makes a local optimal solution yet may not be the global optimal solution.

For example, assume that we have two data reallocation options. Reallocating variable a and b gives the equal amount memory space usage, but variable a is more frequently used than b. The UCC-DA is more likely to reallocate variable b instead of a, because this decision will cause less code updates. However, if later the UCC-RA decides to assign another register for a, then these a related instructions still have to be updated. It is more energy efficient to reallocate variable a instead of b while doing UCC-DA.

For this type of cases, the ILP based solution can produce a near-optimal solution, because it solves the UCC-RA and UCC-DA problems at the same time. However, it will take longer time. This is the tradeoff between the two proposed algorithms.

3.2   UCC TECHNIQUES FOR DSP APPLICATIONS

As discussed in Chapter 2, with the address generation unit (AGU), DSP instruction set supports the post-incremental, post-decremental, and pre-decremental instructions, where address calculated can be in parallel with the other operations, if the next memory location to be accessed is within the auto modify range of the previous access. Thus, no extra address calculation instruction is needed for these cases. An optimal data allocation algorithm is desired to allocate the variables accessed sequentially in adjacent memory slots. So that, less instructions are needed to explicitly update the address registers, which will reduce the code size and execution overhead.

Because of such connection between the data allocation and code generation in DSP applications, keeping data allocation similar as the older version will keep the addressing modes similar as the old version, therefore the generated binary image. Thus, an update-conscious data allocation scheme is desired for DSP application updates.

Besides that, how to assign address registers to the variables can also affect the generated binary similarity. Keeping the address register allocation result similar as the old version also improves the generated binary similarity, which reduces the patch size.
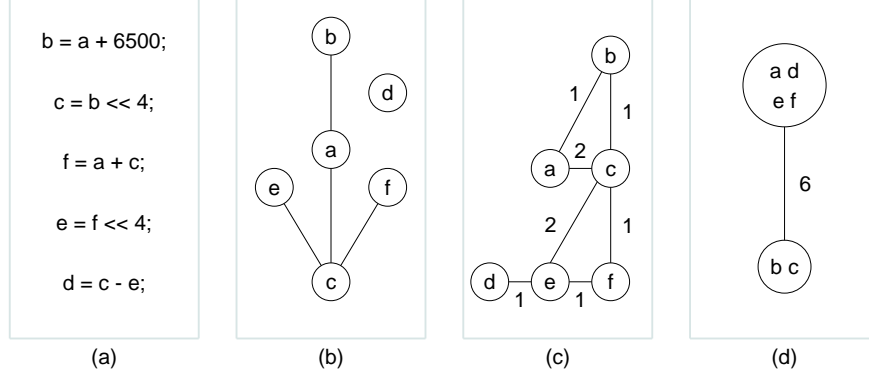
Figure 23: An example of data allocation for DSP applications. (a) The IR code; (b) The access graph; (c) The interference graph; (d) The data allocation result.

3.2.1 Data allocation problem for DSP applications

An motivation of the proposed UCC data allocation (UCC-DA) scheme is shown in Figure 23. When a DSP application undergoes a small update, the code before and after the change are similar. Update oblivious schemes generate the new memory layout and its corresponding binary code without considering the similarity between different versions.

However, an UCC-DA algorithm reads in the old access graph and its interfere graph, and strives to generate a new memory layout that minimizes the update script, i.e. the difference between the new and old binaries. A sensor node only needs to download the update script and regenerates the new binary and/or the new memory layout (Figure 26) with simple interpretation.

Figure 24 illustrates the data allocation result of the example shown in Figure 23 using an UCC-DA algorithm. Figure 24(a) shows the new code after a simple change of 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). Out of seven instructions to be updated in the old code, four of them are due to the data allocation change.
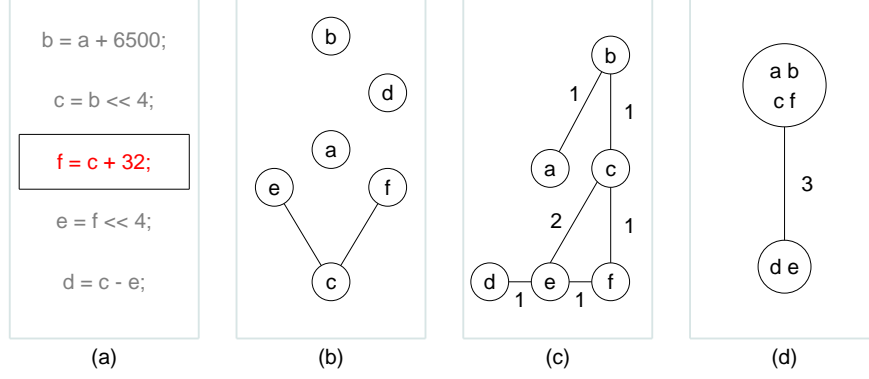
Figure 24: A motivational example for the need of UCC data allocation for DSP applications. (a) The C code after a simple update over Figure 23; (b) The new interference graph; (c) The new access graph; (d) The new data allocation using CSOA, showing significantly different layout from the original assignment shown in Figure 23(d).

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

The design goal of the UCC data allocation is to keep the memory layout similar between the old and new versions. with minimal run-time performance loss. In order to solve this problem, a incremental coalescing offset assignment scheme is proposed. This algorithm assumes that there is only one address register and solves the data allocation problem by keeping the data allocation similar as the old data allocation result.

3.2.2.1 Incremental coalescing single offset assignment (ICSOA)

To minimize the update script, I propose to perform update-conscious code updates through incremental coalescing SOA (ICSOA) (Figure 26). When a DSP application undergoes a small update, the change does not greatly affect the binary code. 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.

The pseudo code of ICSOA is shown in Algorithm 2. It first builds the access graphs

	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 for GCC result is "slot 0: a, b, c, f; slot 1: d, e".

The memory layout for UCC result is "slot 0: a, d, e, f; slot 1: b, c".

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

(a) Server side update conscious offset assignment

(b) Update the code on mobile device
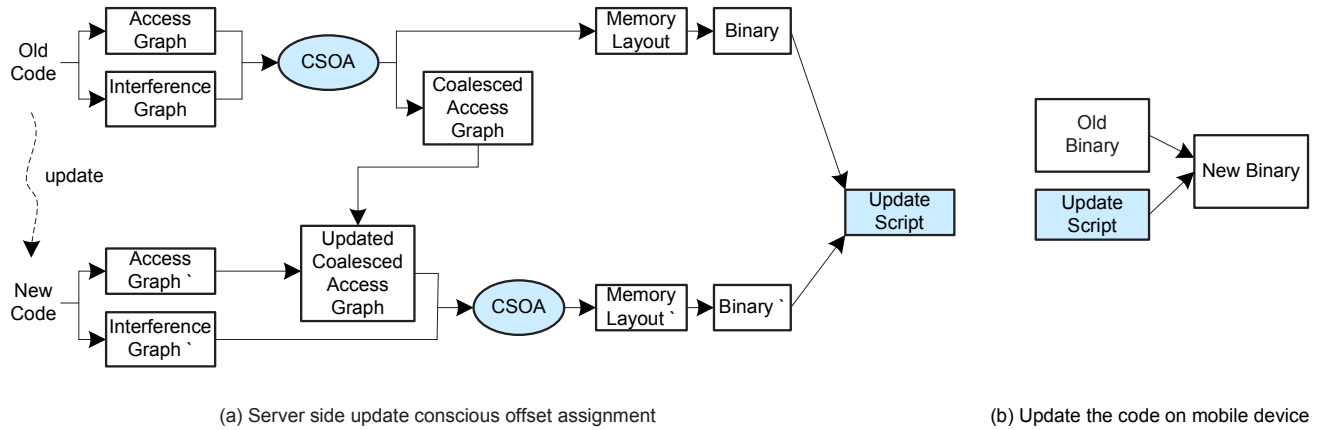
Figure 26: An overview of ICSOA-based code update scheme.

before and after the code update, performs the CSOA algorithm, retrieves the coalesced variable assignment in CAG_1, updates the new access graph AG_2, resolves possible conflicts when applying the old layout to the new code, and calls CSOA again to find the new offset assignment.

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;

It combines the access graph result of the old version (CAG_1) and the newly generated access graph (AG_2), into a new access graph (AG_{NEW}). We build 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 code quality loss. 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 coalesced 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 group nodes will be removed from AG_{NEW}. At the end, the function adjusts the weights of impacted access edges accordingly to finish the update.

In the code update, two variables that were coalesced in the old assignment may interfere with each other. We identify this as a *conflict* and use the function *resolve_conflicts()* to resolve it.

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

$$\frac{Num_{local\_itfs}}{Num_{local\_acs}}.$$

Here, Num_{local_itfs} represents the number of interferences between the variable and the other group members, and Num_{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 one by one until all the interferences in the group are resolved. By doing so, the variables that create more interferences 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 to 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, 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.

3.2.3 Address register allocation and data allocation for DSP applications

In practice, more address registers are available on DSP chips. Keeping the address register assignment to the variables the same as the old version may also improve the binary level similarity. I propose an update-conscious address register allocation scheme for DSP applications here that generates a similar association between the address registers and variables with the old version, and integrate that with the data allocation scheme (ICSOA).
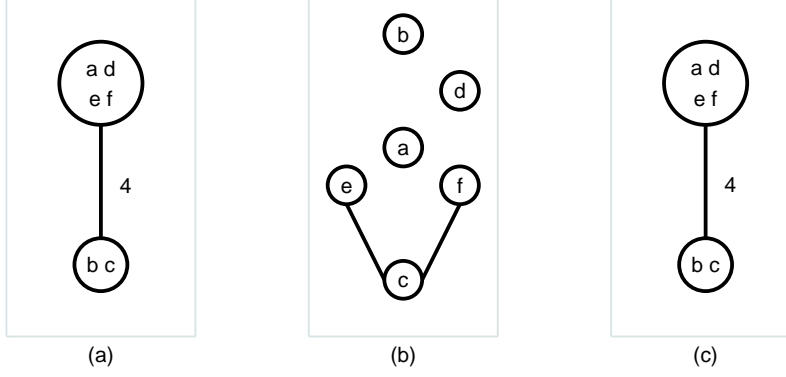
Figure 27: An example of ICSOA scheme: (a) AG_{NEW}, the updated access graph; (b) IG_2, the new interference graph; (c) The final offset assignment.

3.2.3.1 Incremental coalescing general offset assignment (ICGOA)

The ICSOA algorithm proposed above solves the data allocation problem for DSP applications when there is only one available address register, while this ICGOA algorithm here is to solve more realistic problems. It is developed based on the CGOA algorithm [46]. The goal is to solve both address register allocation and data allocation problems for DSP applications.

This problem can be formulated as how to divide the variables into a certain number of groups, assuming that accesses to the variables in the same group will use the same address register. Then we can use the ICSOA algorithm proposed above to allocate the variables in each partition group and generate the complete memory layout by combining the partial data allocation results. The detailed algorithm is presented in Algorithm 3.

It first uses the CGOA algorithm [46] to produce the variable partition of the old version based on the old access graph and interference graph using a heuristic based algorithm. 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 a higher *global interference number* is processed earlier than the ones with lower *global interference numbers*, because they have more constraints. Then, CGOA determines the partition group that the variable belongs to according to the

---

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;
    the number of address registers N_{AR};
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*;

---

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 *local interference number* between each variable and each partition group is calculated and the variable is assigned to the group that has the smallest *local interference number*. 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. New variables are considered next. Same as CGOA, the new variables are first ordered by the *global interference number*. Each variable tends to be assigned with the group that has the fewest *local interferences*. This generated new partition $Partition_2$ should be very similar as the old one, because it just incorporates the variable changes to the old partition.

After that, we run the ICSOA algorithm within each partition group to generate the memory layout for the variables inside each group. The ICSOA results are then combined to form the final result.

4.0 SOFTWARE DIFFERENTIAL PATCHING

With the new binary versions generated by UCC technique, I define a update script format to summarize the code difference between the base binary and the newly generated binary. The framework then will transmit the patch script to the sensors, and let the sensors reconstruct the new binary. As shown in Figure 28, the old version binary E and the new version binary E' are first compared, and then the binary level differences are formatted as the update script U. After the sensors receive the complete U, they will retrieve the new binary image E' by combining U with the old binary image E which already exists in the sensor memory.
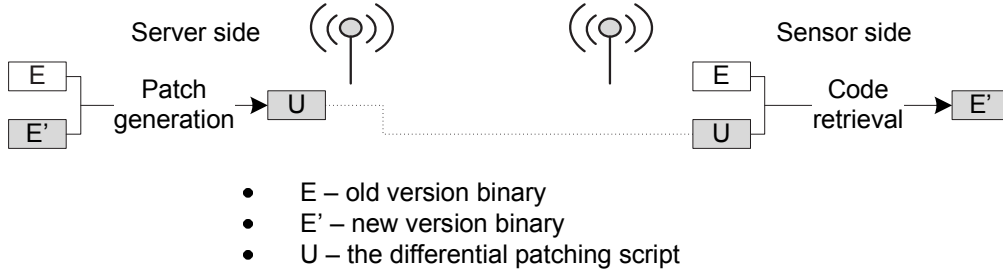
Figure 28: Patch generation and binary reconstruction.

The binary code may be changed due to functionality change or data layout change, thus, I separate these two kinds of changes in the update script, as the functional script part and data layout part representatively. 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 adopted four simple code update primitives from the prior work [51], and propose three advanced functional primitives and three data layout primitives to describe the higher level code changes.

4.1 INSTRUCTION BASED PATCHING

I use the functional binary update primitives to describe the functional changes, such as adding, removing or updating instructions caused by functionality changes. I adopted the simple primitives from the prior work [51] and proposed the advanced primitives to solve more complicated code compression problems. The difference between the advanced primitives and the simple primitives is that they are not used to describe the simple bit level comparison results, but higher level structure changes such as the destination address shifting for a group of instructions.

The format of the script primitives is shown in Figure 29.
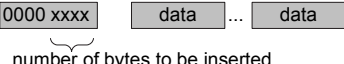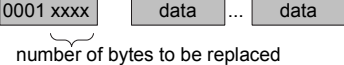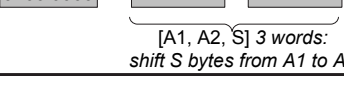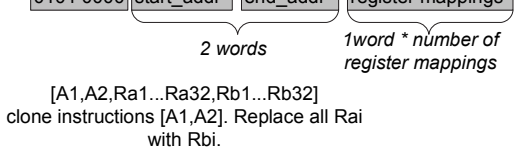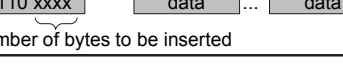
Primitive	Format and Operation	Size (bytes)
insert	0000 xxxx data ... data number of bytes to be inserted	1 + number
replace	0001 xxxx data ... data number of bytes to be replaced	1 + number
copy	0010 xxxx number of bytes to be copied	1
remove	0011 xxxx number of bytes to be removed	1
shift	0100 0000 [A1, A2, S] *3 words:* *shift S bytes from A1 to A2*	7
clone	0101 0000 start_addr end_addr register mappings *2 words* *1word * number of register mappings* [A1,A2,Ra1...Ra32,Rb1...Rb32] clone instructions [A1,A2]. Replace all Rai with Rbi.	5 + 2*number
insert_access	0110 xxxx data ... data number of bytes to be inserted	1+number

Figure 29: The functional patch script primitives

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 incor-

porated. The `copy` and `remove` primitives take one byte each and specify the size of old data/instruction block to be copied or removed.
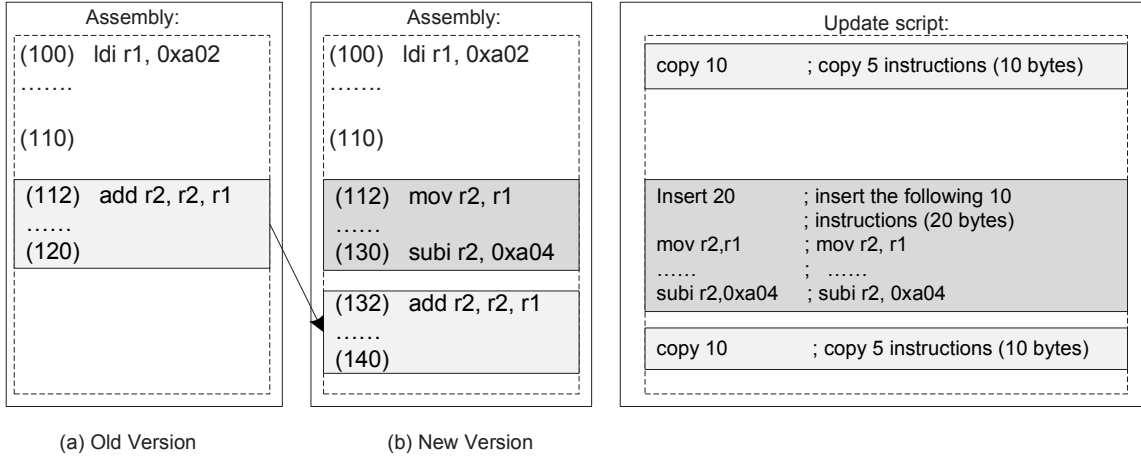
Figure 30: An example of the simple primitives. New code `[112,130]` is inserted. `[112,120]` in the original code is now moved to `[130,140]` in the new version.

Figure 30 shows an example of the simple primitives. The new version contains three chunks of code, `[100,110]`, `[112,130]`, and `[132,140]`. Both the first and third chunks can be found in the old code while the second chunk is new. Therefore the update script contains two `copy` primitives and one `insert` primitive. The `insert` primitive has a one-byte opcode and ten 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 has been generated in the new binary image. The `insert` primitive inserts the instructions in its data part into the new binary image, and moves the pointer in the new code to the end. 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 reads the instructions from the old binary, and moves both pointers.

4.1.2 Advanced primitives

In the experiment, I observed some code structure changes that affect more than one instruction. For example, when the register assignment of one variable is different in the new binary, all the instructions that access this variable need to be updated. Since the affected instructions are usually more than one, it is cheaper to incorporate such register assignment changes in the patch script, other than the binary level differences. Based on this observation, I propose three advanced primitives in my design.

4.1.2.1 shift

As some code may be inserted into or removed from the base binary in software update, the absolute address of the instructions may be changed in the update. Such change might cause the destination address changes for branch instructions. I use the `shift` primitive [51] that informs the sensors about the destination address shifts, so that the sensors can incorporate such code changes on side. Instead of explicitly updating all the affected branch instructions using several `update` primitives, now we can use one `shift` primitive to describes such code changes. The update script size can be significantly reduced.

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 in the range `[A1,A2]`, will have the destination addresses shifted by `S` on the sensor side.

ability — it picks up each control transfer instruction, checks whether its target address falls in the range that needs to be shifted, and updates them with new addresses. branch instructions, their target addresses can be computed by adding the relative offset to the address of the current instruction.

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.

Assembly:
(100) ldi r1, 0xa02
…….
(106) cpi r1, 0xa04
(108) brne .+2
(110) ldi r1, 0xa04
(112) add r2, r2, r1
……
(120)

Assembly:
(100) ldi r1, 0xa02
…….
(106) cpi r1, 0xa04
(108) brne .+2
(110) ldi r1, 0xa04
(112) mov r2, r1
……
(130)
(132) add r2, r2, r1
……
(140)

| Update script:	
shift 112,132,10	**;block [112,120] is moved to ;[132,140].**
copy 10	; copy 5 instructions (10 bytes)
Insert 20	; insert the following 10
mov r2, r1	; instructions (20 bytes)
……	; mov r2, r1
	;…...
copy 10;	; copy 5 instructions (10 bytes)

(a) Old Version   (b) New Version

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.

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 on the sensor side such that it can decode each instruction type to extract the desired target address.

The sensors will interpret the `shift` primitive in the following way. For the absolute branch instructions, the original destination is decoded first and if it falls in the shift range, it will be updated according to the offset encoded in the primitive. For the relative branch instructions, their target addresses can be computed by adding the relative offset to the address of the current instruction.

It is implemented by maintaining a shift information table. When encountering a `shift` primitive, one shift entry is added to the table, which includes the start address, end address and shift offset. When copying one instruction from the old binary to the new binary, the interpreter will check this table to see if it is a branch or jump instruction. If the target address falls in any shifting range, the destination of this instruction will be updated.

4.1.2.2 clone

In the experiments, I found that the binary code of the `inline` functions called at different locations looks very similar with each other, although different register are used. This is because they are compiled from the same source code, however, due to the different context, different register assignment decisions may be made. Although they only differ in the register usages, when an `inline` function is introduced or updated, similar copies are inserted or updated more than once in the binary, which is a waste.

Based on this observation, I introduced 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 block `[A1,A2]` and `[B1,B2]`. The patch generator uses block `[A1,A2]` as the comparison base, and tries to match the register allocation between these two blocks. Assume the register mapping between them is shown as below, $(R_{a1}, R_{a2}, ..., R_{an}) \Rightarrow (R'_{b1}, R'_{b2}, ..., R'_{bn})$. Given such information, instead of using a sequence of the simple primitives to describe the updated/new code of block `[B1,B2]`, we could rather copy instructions from block `[A1,A2]`, and slightly change the register assignments according to the register mapping to rebuild block `[B1,B2]`.

As shown in Figure 29,The `clone` primitive has one-byte opcode, and another several bytes to specify the starting and ending address 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. The instance that is stored with the lowest addresses will be considered as the master copy. The other instances will 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

57

script for [200,206] by using the `insert` primitive, the `clone` primitive is used to specify that the second code block clones the block [100,106] while registers r_1 and r_2 needs to be updated to be r_{11} and r_{12} respectively.

Assembly:	Assembly:	Update script:
(100) ldi r1, 0xa02 (102) ldi r2, 0xa04 (104) add r2, r2, r1 (106) std 0xa04, r2 …...	(100) ldi r1, 0xa02 (102) ldi r2, 0xa04 (104) add r2, r2, r1 (106) std 0xa04, r2 …... (200) ldi r11, 0xa02 (202) ldi r12, 0xa04 (204) add r12, r12, r11 (206) std 0xa04, r12	copy 100 ; copy 50 instructions (100 bytes) clone, 100, 4, (r1,r2), (r11,r12) ; ; clone 4 instructions starting from address 100 ; replace register r1 with r11 replace register r2 with ; ; r12

(a) Old Version   (b) New Version

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

The `clone` primitive can reduce the script size when the `inline` function is called at multiple places, and the register mapping is clear. However, if the register mapping is too complicated the script size could be very big, in which case it is better to use the simple primitives, such as `insert` and `replace`. In addition, it requires the sensor-side interpreter to have simple decoding ability to extract register names from different instruction types, and replace them with new ones. The sensors need to reconstruct the code segment by replacing the registers in the master copy according to the patch script. Each clone operation will need to decode the instructions in the master copy and replace the registers. Thus, when the master copy is frequently cloned, it is more efficient to store the master copy in a storage buffer and add tags to each instruction indicating whether this is a memory access instruction and which register is used, so that the clone process can speed up.

4.1.2.3   insert_access

When inserting a new memory access between two existing accesses, we may need 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 it is to include the memory address difference in the script and let the mobile devices generate the correct addressing modes for the related instructions. Thus, I introduce an `advanced primitive` – `insert_access`.

(a) Memory layout

(b) Old Version

(c) New Version

(d) Script w/ simple primitives

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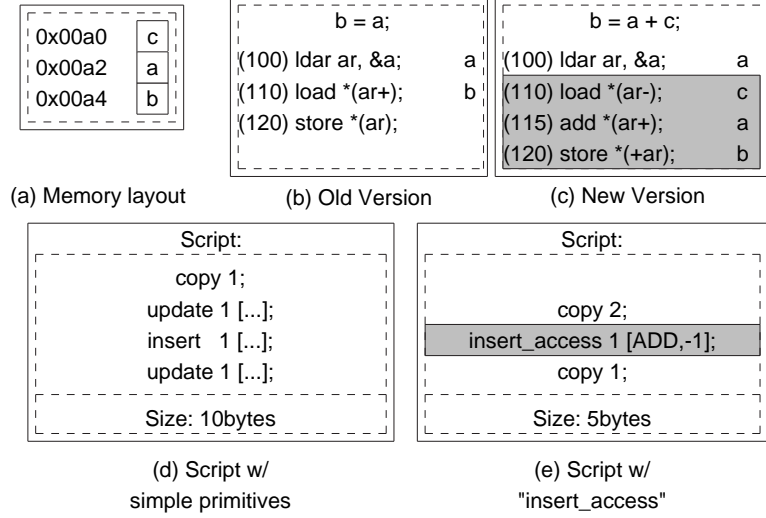
Figure 33: An example of the `insert_access` primitive: (a) The memory layout for both versions; (b) The source and assembly before code update; (c) The source and assembly after code update; (d) The update script using the simple primitives; (e) The update script using the `insert_access` primitives.

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

$$[operation, \delta_{diff}]$$

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

$$\texttt{insert\_access} \quad 1 \quad [ADD, -1].$$

59

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` primitives allows the sensors to correct the addressing modes before and after the newly inserted memory access.

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`. Based on these two instructions, 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.
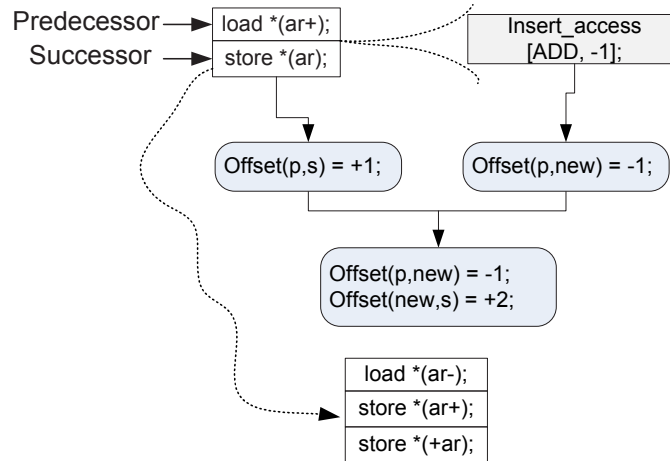
Figure 34: An example of the interpretation procedure of the `insert_access` primitive .

Figure 34 demonstrates the interpretation procedure of the example given 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, 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 interpretation for functional primitives

The received patch scripts will be stored in the program memory. When the script download is complete, the sensors will run a simple script interpreter to incrementally reconstruct the new binary image. The reconstruction is based on the received the patch script and the old binary image which is stored in the program flash on the sensors. The generated new binary image will be stored in the program flash as well. When the primitive interpretation is complete, the sensors will load the new binary image back to the program memory and restart to execute the new version. The old image will be kept in the program flash until the space is needed to store newer versions, so that when an execution error happens, the sensors can roll back to the older version. However, because sensors use cyclic redundancy check (CRC) to ensure the data integrity while transmitting the patch messages, and the reconstruction has been tested on the server side to make sure that it can generate the correct binary image, there is a very small chance that the new binary image is not functioning correctly.

The flash memory used to store both the old and new binary images can be read in a random access fashion, so the pointer that is pointing to the old binary can be moved arbitrarily when it is needed. However, one limitation of the flash memory is that it has to be programmed at block levels, e.g. 256 bytes on for mica2 sensors [20]. In order to change one byte, the sensor has to read the correspond block into program memory, modify it and then write it back. Thus, the constructed new binary needs to be buffered in the program memory first until it reaches the size of a flash block, then the code block will be written back to the program flash. The size of the temporary code buffer should be a multiplier of the block size.

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

To interpret the `insert` and `replace` primitives, it copies the instructions from the data part of the primitive to the new binary. To interpret the `copy` primitive, it copies 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 image. The pointer in the old binary is shifted according to the number of bytes that have been copied, removed or updated, according to the script.

When encountering the `shift` primitive, one entry that records the start address, end address and shift offset is inserted to the shift table. To interpret the `clone` primitive, the master copy will be read from the program memory and register usage will be modified to construct the cloned copy. The master copy needs to be read 0 to K times, where K is the number of cloned copies that it has. In order to avoid reading the program memory K times, this master copy can be buffered in the program memory. The `insert_access` primitive will decode the predecessor and the successor to correct the addressing mode. Because the generated code is first buffered in program memory, and there is usually one or two instructions inserted by this primitive, both the predecessor and successor may still in the temporary code buffer. Thus, this operation may not cause read or write to the new binary image.

When the whole code construction process is complete, the new binary image will be copied to the program memory from the program flash. The sensor will then restart to run the new code.

The algorithm shown in Algorithm 5 is called whenever a 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 will be updated for the address shift. When the temporary buffer is full, the code block will be copied to the program flash.

The memory space required for the interpreter include 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 4 kbytes, the start address and end address can be encoded using 3 bytes. The shift offset can encoded using 1 byte. Thus, the storage required for each entry is 4 bytes.

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** (; $P_S \neq script.end()$; $P_S = script.next\_primitive()$) **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 += replace_bytes(P_S)
9:     **break**
10:    **case** copy:
11:     **write_code_buffer**(P_N, P_O, copy_bytes(P_S))
12:     P_O += copy_bytes(P_S)
13:     **break**
14:    **case** remove:
15:     P_O += 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 $i =$ form_inst(opcode(P_S),*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

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

I observed that binary changes at several places may be caused by one memory layout change in the experiments. Assuming variable a appears in several places in the code and is relocated to a new memory location, we may generate a script with multiple update primitives each of which presents one instruction level change. Instead, if the script interpreter on mobile devices can decode DSP instructions, and identify all the uses of a, it is possible to send one "relocate a" primitive instead of individual instruction update.

Let us call the binary instructions that are inserted, removed, or changed due to the offset assignment changes as *addressing mode change* (AMC) instructions. The motivation of developing *data primitives* is to reduce the transmission of AMC instructions, and let the mobile devices construct them by themselves. Compared to the *insert_access* primitive, data primitives are designed to update the code in more than one place.

Primitive	Format and Operation	Size (bytes)
copy_slot	`0111 xxxx` number of data slots to be copied	1
insert_slot	`1000 xxxx` `variable` ... `variable` number of data slots to be inserted	1+number
shift_slot	`1001 xxxx` `start_slot` `offset` number of consecutive unchanged slots	3

Figure 35: The data layout patch script primitives.

4.2.1 Data update primitives

In order to update the AMC instructions automatically, the offset assignment changes (rather than affected instructions) need to be transmitted.

Figure 35 lists the *data* layout change primitives that are used to specify the memory layout change. I only consider the allocation of scalar variables here. Each memory location contains one variable or multiple coalesced variables ([46, 62, 63]). The old memory layout is maintained on the sensor, and when it receives a update patch with the data layout changes, it will reconstruct the new memory layout onsite. Both the old and the new memory layout are maintained as tables. Each row represents the variables that share the same memory slot and the rows are ordered by the address of the corresponding memory slot.

4.2.1.1 copy_slot

Similar as the `copy` primitive in the functional primitive set, this primitive copies multiple memory slots from the old memory layout to construct the new memory layout. There are two pointers pointing to the active memory slot in the new and old table respectively to accomplish the interpretation of this primitive.

4.2.1.2 insert_var

This 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 location, which is omitted to keep the script compact.

4.2.1.3 shift_slot

This 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 that need 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 update primitives* to generate the new memory layout, and then interprets the *functional update primitives* to construct basic blocks by inserting, removing, or updating certain instructions on top of the old binary version. 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.

However, it may require additional information to fix the addressing modes on the mobile device side. As shown in Figure 36, CSOA coalesces multiple variables — both `a` and `e`, in one memory location `0x00a2`, a code update may re-allocate `e` to `0x00a0` while keeping `a` in the same memory slot. This complicates the code update as some accesses to `0x00a0` should be updated while others should not.

Figure 36 illustrates my solution to 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 to the re-allocation.

To update the addressing mode in the new code, a query is sent to the coalesced variable
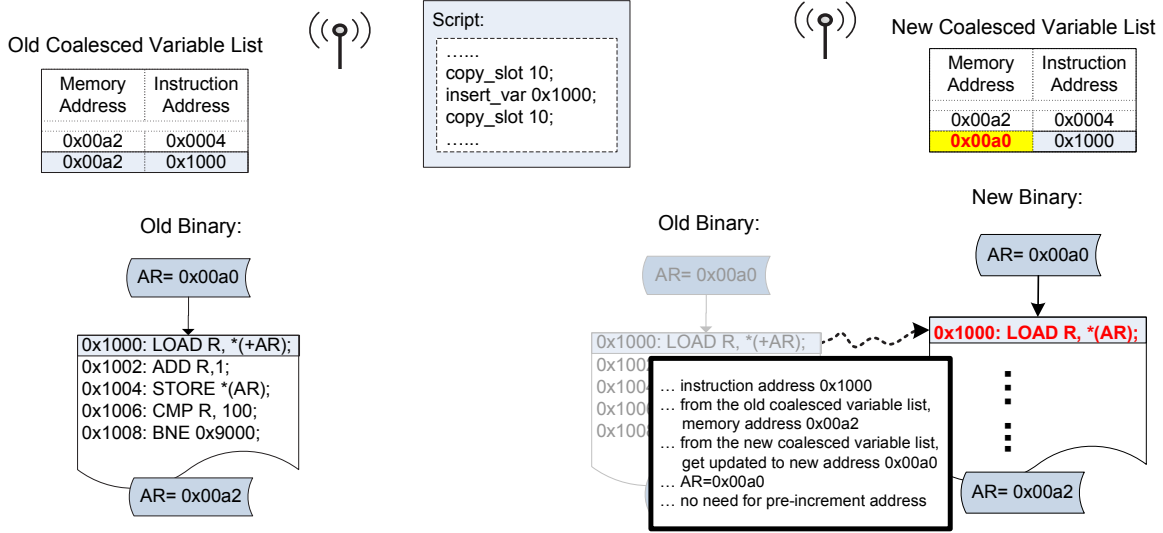
Figure 36: The code construction procedure of the data primitives. The left shows the server side, and the right shows the mobile device side updates).

list, from which we know 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 and ensure the exiting AR contains `0x00a2`.

From this discussion, the interpreter needs the following information to fix the addressing modes:

- A coalesced variable list to distinguish each of 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. `a` is assigned to a different memory location, we need 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 range 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, variable `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 its 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 mobile devices before deployment. The updates to the coalesced variable list is transmitted with the code update script. The coalesced variable list update primitives will be discussed later.

AR in/out value list. As discussed before, we need the AR in and out values for each basic block in order to generate the correct addressing modes on the mobile device side. 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, ending addresses, the address register's entering, 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 mobile devices 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 mobile device side, the server explicitly sends out the AMC instructions that follow an inserted/updated/removed instruction, and those that are the last instruction of a basic block.

Complexity analysis. The following pseudo code Algorithm 6 presents the algorithm that is used to correct the addressing modes based on the data change 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 do 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, so they can both fit in the program memory for fast access.

69

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

Input: Instruction i which will be copied from 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

As introduced in Chapter 1, there are two circumstances in software update, software upgrade and software switch. The software upgrade happens when the application that is already running in the WSN needs to be changed for bug fix or adding new features. So in this case, there is one source node – the sink, and multiple destination nodes in the network. However, The software switch happens in MA-WSNs, where multiple applications are already deployed in the network. Because some neighboring nodes may already have the wanted binary code in memory, the code distribution problem here is how to route the code image from sensors to sensors. In this case, there are multiple source nodes in the network, which is different from the software upgrade case. Because of this difference, I propose to use two different code distribution protocols for the two different cases.

5.1 BROADCAST BASED CODE DISTRIBUTION PROTOCOL (DELUGE)

While doing software upgrade, the upgrade patches are generated on the sink node using the proposed update-conscious compiler techniques to improve the code similarity with the older version of such application. Then the patch is generated in the script format as mentioned above.After the patches are generated on the sink node, the network protocol Deluge [32] is used to disseminate the scripts to the network.

The protocol works as follows. First, the whole update script is divided into fixed size pages. At the beginning, the states of nodes are set to *Maintain* state. Each sensor node keeps broadcasting the advertisement messages (ADV) periodically, which contains the information of the application code that it has. When a sensor node S receives an advertisement, which

71

indicates that the neighbor N has a newer version of the application in its memory or has finished downloading more pages, node S will send out a request message (REQ) to N to request for a 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 will change state to *Transmit* when it receives the request message from S. Then it will start sending all the packets of the requested page to node S. After the code transmission is finished, the state will be changed back to *Maintain* state. Figure 39 shows the advertise-request-data handshaking protocol used Deluge [32].
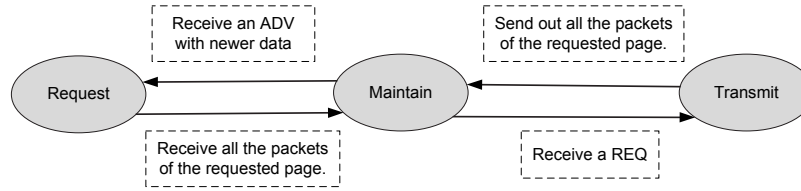
Figure 39: Advertise-request-data handshaking protocol in Deluge.

These packets may be encrypted and/or authenticated for security protection [27, 37]. The packets may also be grouped so that when remote sensors receive groups out of order, they are still able to perform updates independent of the receiving order.

5.2 MULTICAST-BASED CODE REDISTRIBUTION PROTOCOL (MCP)

5.2.1 The software switch problem in MA-WSNs

While doing software switch, the problem is a little bit different. The following example shown in Figure 40 illustrates the protocol design challenges here. 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.

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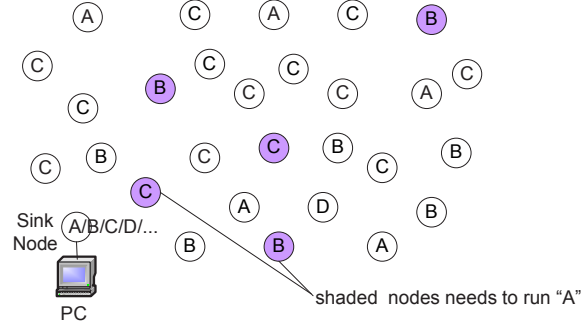
Figure 40: An example of software switch in a multi-application WSN (MA-WSN).

to unnecessary packets transmissions to the nodes that don't need it. The other solution is to let requesting nodes initiate code dissemination and fetch A from nearby sensors. Melete [61] 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 create significant signal collision.

5.2.2 A multicast-based code redistribution protocol (MCP)

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] scheme discussed above, the data messages are only multicast to the requesters, which avoids the unnecessary packet transmission in the network. With the guide of AIT, the request messages can be directly sent to the source nodes, which avoids the request message flooding in the network.

An overview of this protocol is as follows.

73

- 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 its overheard ADV messages in an *application information table (AIT)*.

- 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]" 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 followings.

  (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*.

- A *requester* periodically sends out requests (i.e., REQ messages) to its closest source, until it acquires all the pages of application A. Instead of broadcast, the REQ messages are sent to the source via multicast. A requester resends the REQ message until it timeouts. It tries to request data from each source node several times before marking the node as a *temporary non-available* source.

- 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 has 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).

5.2.3 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). Fig. 41 illustrates the algorithm.

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 two closest source sensors — the source ID and number of hops to the source (S, H); (iv) the CRC checksum. If a sensor has multiple known applications, it advertises them in a round-robin fashion. Note that a sensor may not have the code images of all its known applications.

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

```
n1        n2        n3

n4        n5        n6

n7        n8        n9
```
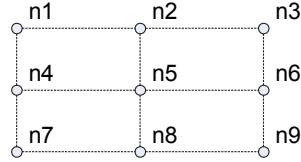
On Node N9:

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).

The AIT 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. The size of each application entry in the AIT is 12 bytes. Assume that the number of the applications running in the network is 10, the AIT size will be only 120 bytes, which makes it fit perfectly in the program memory.

When an incoming ADV message contains new information, the corresponding entry in

the AIT is updated. Assume a sensor S1 receives an ADV message from S2, and the message identifies two nearby sources (S3, H3) and (S4, H4) where H3 and H4 indicate the number of hops from S2 to sources S3 and S4. If S1 already records the information of three sources (S5, H5, U5), (S6, H6, U6), and (S7, H7, U7), then it updates the AIT table according to the following rules.

- If one entry in AIT table records the previous message from the same uplink S2 and it refers to the same source, e.g. S5=S3 and U5=S2, 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. S5=S3, U5≠S2, and H5>(H3+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.

Each sensor advertises the application 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 we elaborate next.

5.2.4 Request multicasting

In MCP, a requester continues to send out request messages until it receives all pages of the target application. Given the target application, the requester searches the AIT for a closeby live source and constructs a REQ message as follows

$$REQ = [S, H, pgNum, bv]$$

where S indicates the selected source node, H indicates the maximum number of hops that the message may travel, pgNum and bv indicate the current working page and the requested packets in the page. If the AIT records more than one source node, then the requester selects the closest live source and sets H to h+δ where h is the number of hops to S (recorded in the AIT), and δ is the hop count slack allowed in the dissemination. Fig. 42 illustrates the involved nodes when h=2 and δ=1. These nodes routed through a gradient-based region [17] to the source.
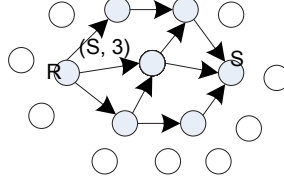
Figure 42: Gradient-based request routing. R and S are requester and source nodes respectively; h=2; δ=1.

A requester continues sending the REQ messages when it can not finish the page before timeout. After several tries, it marks the source that it tried to reach as an *unreachable* source. The number of tries varies based on the distance to the source.

If the AIT does not record any nearby source, then the requester sets S to be *null*, indicating the REQ message is sent to all neighbors. After receiving a broadcast request, an idle forwarder forwards the request unless the message has travelled the maximum number of hops; an idle source node always responds with requested packets.

from which the request was

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 who is currently working on page y, and $x + 1 < y$, then the node quits the current state and switches to serve the request. If the node is a forwarder, then it forwards the request; if the node 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.

5.2.5 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 work on large-number-indexed pages, and a forwarder has a high possibility to receive requests for these pages.

To improve transmission 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 is slow and writing it has to be performed in 256-byte blocks, which complicates the design and wastes the energy.

Caching on a requester is straightforward as the sensor always 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 serves these requests. In MCP, a forwarder gives priority to pages with smaller indices. We set a timer for the cached page and clear the page after serving a request or timeout.

5.3 SIMULTANEOUS CODE DISSEMINATION

As shown in [61], different applications in a MA-WSN usually share some code segments. For example, two applications may be designed for sensing and processing two different events like wildfire and animal mitigation. While the data processing components are different, the routing code could be similar. If one application has already been installed on some sensors, then at the time when a remote sensor wants to load the other application, it is energy efficient to fetch the common code from these peer sensors instead of the sink.

Fetching code from peer sensors exhibits two advantages: (i) remote requesting sensors (i.e. the sensors that need to switch their running application to the new one) can start early and fetch the code in parallel without waiting for the progressive code dissemination

from the sink. (ii) since only a subset of sensors get involved in dissemination, the message overhead can be greatly reduced. Without losing generality, we assume A and B share S_{ab} common packets while A and C do not share any packet. When a sensor needs to switch to run application A, it fetches shared code from nodes that have B and the rest of the code from the sink.
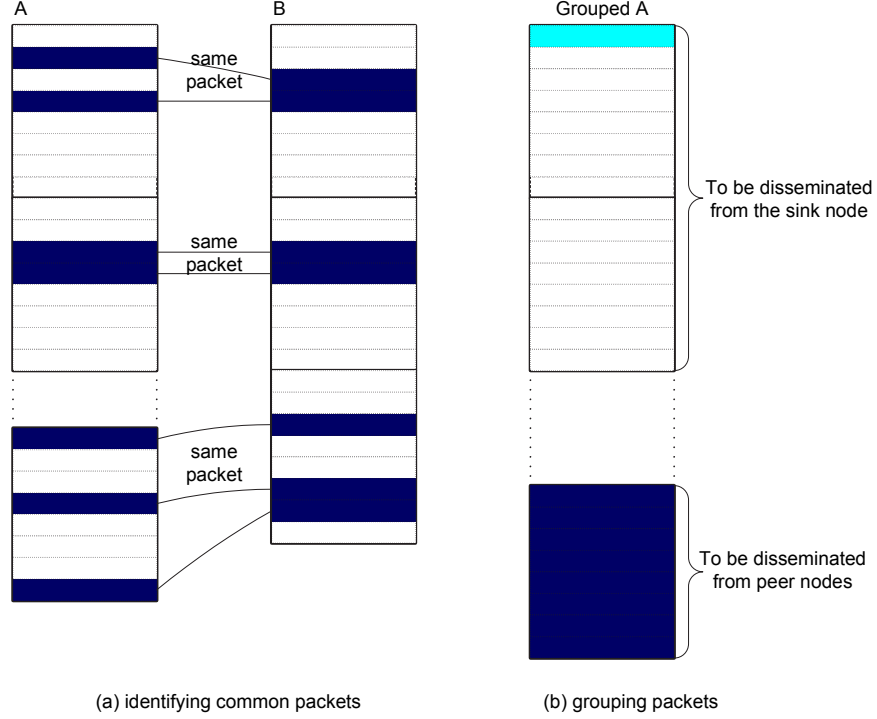
Figure 43: Simultaneous code dissemination.

The basic dissemination unit in MA-WSN is a packet that contains 23 bytes payload, similar as that in the default multi-hop dissemination protocol Deluge [32] in TinyOS. To enable code dissemination of two types of packets, the code segments needs to be reorganized, as shown in Fig 43. Given the above application A to be disseminated, we first divide it into a sequence of code packets, and mark all packets that are shared with B. We compare at the packet level in this paper while techniques have been proposed to compare two applications and generate difference at different levels [43, 61] After marking the code, we group packets based on if they are marked or not, and add two bit vectors (one vector per application and one bit per packet) to guide the code reorganization. The bit vector for application A (or B) indicates the locations of marked packets in application A (or B). For example, a bit vector

6.0   EXPERIMENTAL RESULTS

In this chapter, I will present the experimental results. First, the methodology that I used to construct the sensor software update benchmarks are addressed. Then, I divided the experimental results into two parts, the data collected before code dissemination and the data collected after code dissemination. The former part compares the trade-off between the generated script size and the code performance for both general purpose and DSP applications. The latter part compares the network traffic and time consumption between the proposed protocol with the other existing schemes.

6.1   BENCHMARKS

Because the software update management is a new problem to study in the wireless embedded system community, there is no existing update benchmarks available to evaluate the performance gain of my proposed framework. Therefore, I built a sensor software update benchmark suite which covers the software update cases for both general purpose applications and DSP applications that run on the wireless embedded devices.

The base benchmarks are from the general purpose sensing applications included in TinyOS [5]– an open-source operating system designed for wireless embedded sensor networks, CryptoLib – the encryption library, and the DSP applications included in DSPstone [1] benchmark suite. Upon the base benchmark, I created the pairing update benchmark using three different techniques and categorized them based on the update levels.

6.1.1 Update levels

There are three update levels according to their impact on code structures: (a) low level updates, which are made to local basic blocks; (b) medium level updates, which include changes in a large function or across several functions, but still preserve the overall structure of the original code; (c) high level updates, which significantly change the code structure. Frequent updates such as code fixes and sensor reconfigurations are mainly low or medium level changes, while replacing the application with a new one introduces medium to high level changes.

6.1.2 Real update benchmarks (real-benches)

One application may have multiple versions that exist in the base benchmarks to add new features of fix bugs that exist in the old version. For example, TinyOS-1.x has over 15 versions, and in each release the enclosed applications may be updated. Updates made to the applications vary from adding one statement to completely reconstructing the code. As the proposed software update management framework targets at low or medium level code updates, I selected the proper sized real code update cases that exist in the base benchmarks as the real update benchmarks. These real update benchmarks will show the real update patterns for the applications running on wireless embedded devices and become the base of generating the other two update benchmarks.

6.1.2.1 General purpose application update benchmark

Figure 44 shows one real update case of the general purpose software. The function unit Deluge [32], a reliable code dissemination protocol enclosed in TinyOS is studied.I studied updates of Deluge from TinyOS version 1.52 to version 1.58. The update details are shown in the figure.

Case#	Versions	Update Level	Update details
R-G-1	1.52 ⇒ 1.53	Low	Add one statement to reset one variable.
R-G-2	1.53 ⇒ 1.54	High	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 ⇒ 1.55	Medium	Modify the condition of two "if" statements.
R-G-4	1.55 ⇒ 1.56	High	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 ⇒ 1.57	Medium	Move two "memcpy" statements to be next to the relative "if" statements.
R-G-6	1.57 ⇒ 1.58	High	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⇒matrix2.c	medium	Move two iterations out of the loop.
R-D-2	speed_control 1 ⇒2	medium	Seven temporary variables are introduced to hold the value of the comparison results.
R-D-3	speed_control 2 ⇒3	high	Multiple global variables are combined into a structure. The reference to the variables are changed due to this change.

Figure 45: Real DSP application update benchmark.

6.1.2.2 DSP application update benchmark

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

6.1.3 Manually generated update benchmarks (man-benches)

Software updates are manually inserted to the base benchmarks to create the manually generated update benchmarks. The manual inserted code is designed to cover all possible code update circumstances including variable insertion/deletion, instructions insertion/deletion/update inside and outside loops, and the control flow insertion/deletion/update.

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 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.

Figure 46: Base benchmarks for general purpose applications.

6.1.3.1 General purpose application update benchmark

The TinyOS application shown in Figure 46 are selected as the base benchmarks to create the manaually generated general purpose software update benchmark.

Figure 47 summarizes the synthetic updates made to these benchmarks. The updates vary from low level, through medium level, to high level changes.

6.1.3.2 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 implement *speed_control* as the real DSP update benchmarks. The detailed benchmarks are listed in Figure 48.

6.1.4 Automatically generated update benchmarks (auto-benches)

In order to study more general cases and to evaluate the compiler performance, I also wrote a tool to generate the update benchmarks automatically.

6.1.4.1 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 the ILP problem created by update-conscious compiler. Because the number of decision variables, constrains 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 that affects to the problem complexity.

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 variables 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 47: 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 on 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 on basic block. **Over ten interferences** are updated.

Figure 48: Manually DSP application update benchmark.

6.1.4.2 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, to cover broader cases. 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.

6.1.4.3 Methodology used to generate the auto-benches

Although I built the auto-benches for general purpose applications and DSP applications based on different input, the methodology used to generate the auto-benches is the same. For the general purpose applications, the IRs of a base application act as the input of the auto-bench generator. For the DSP applications, the access sequence as well as the interferences are the inputs instead. To simply the description, let us define one input statement as one IR statement for the first type of input, one variable in the access sequence or one interference between two variables for second type of input.

For a given input, the point between each two statements and the point after the last statement are called an update point. Given the update percentage, the number of statements that need to be updated is fixed for one base application. A random set of the update points will be selected for updates according to this number. Several update options can be made at each update point, such as inserting a new statement, removing the prior statement or updating the prior statement. A random decision will be made at each update point to create a change.

Several trails are run to create a set of the auto-bench for a given update percentage. There may be some cases that are not semantically correct. For example, the updates that are randomly inserted to the IRs of a general purpose application may not make sensor so that the ILP cannot compute a valid result. These cases are removed from the set and a new valid case will be generated to replace it.

6.2 PRE-DISSEMINATION PERFORMANCE EVALUATION

I implemented my proposed update-conscious schemes, including the register allocation (UCC-RA), data allocation (UCC-DA) and the integrated scheme. I compared the compiler performance between the update-conscious compiler and the GNC C compiler (GCC-RA and GCC-DA representatively). The UCC-RA scheme trades off the run time code performance for a smaller update script that results in a lower transmission energy.

In this section, I will discuss my experimental settings and present the results on code quality, energy efficiency, and compilation time.

6.2.1 General purpose software update 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 manual generated general purpose benchmarks (M-G-1 $\sim$ M-G-13) list in Figure 47 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 [20] running TinyOS [5], an open source operating system designed for WSNs. The processor that Mica2 (MPR400CB model) uses is the AMTEL AVR micro controller — ATmega128L [6].

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

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 [57]). I then integrated the energy model and execution profiles to study the energy consumption tradeoffs with different compilation approaches.

The update script generator is implemented over Diffutils [GNU] to format the output in the proposed format.

6.2.1.2   The generate script size

Figure 47 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 test cases cover a wide range of changes 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 report the changes in the functions that

are directly affected (rather than, for instance, code shifting due to expansion/shrinkage of neighbor functions). In addition, I observed minimal inter-procedural correlation. For example, the same global variable can be assigned with different registers in different functions. Therefore the overall impact of high level updates can be estimated by summarizing the changes in 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 the old binaries. This is the lower bound of existing *binary-diff*-based code dissemination algorithms [51, 59]. That is, I compared my results against the best possible implementation of existing update-unconscious approaches [51, 59].
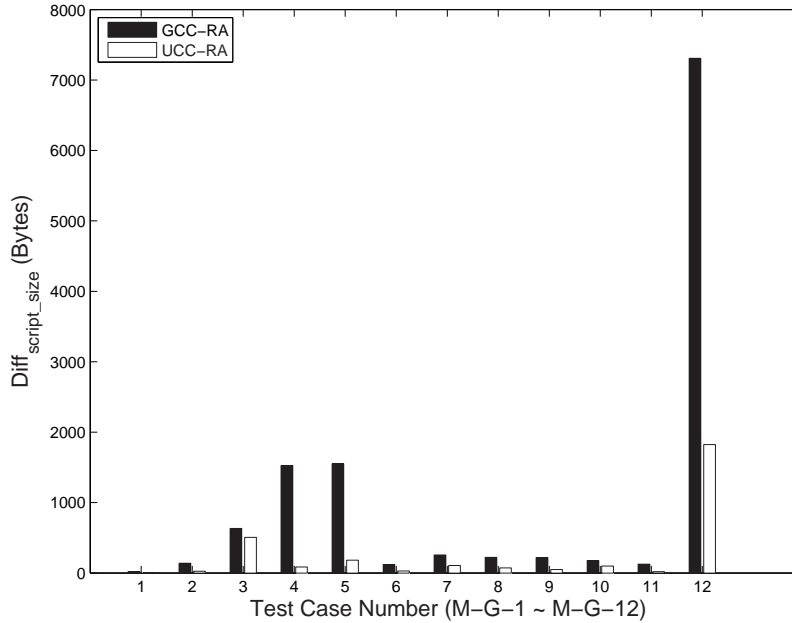

Figure 49: Script size comparison between UCC-RA and GCC-RA.

Figure 49 shows the results, in $Diff_{script\_size}$, for update test cases M-G-1 $\sim$ M-G-12. As I can see, UCC-RA greatly reduces the code difference as it effectively localizes the code changes — the majority of the code can be kept the same. On the contrary, GCC-RA may

generate only local changes (test case 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. Test case 12 introduces several new functions most of which are small *inline* functions. They disturb the register selection in a large function and introduce significant number of differences, 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, 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 need 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`).

6.2.1.3   The generated code quality

Next, 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 version of the binary. This metric also indicates the slowdown in execution time after applying update-conscious compilation.

Figure 50 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 need 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 and achieve overall energy efficiency.

The slowdown from applying UCC-RA is negligible in nearly all cases. For example, the three cycles introduced by UCC-RA in test case M-G-12 accounts for less than 0.01% of 244K cycles — the total number of cycles per single run for the application `CntToRfm`. We study its energy consumption over a long period after many invocations, in the next section.
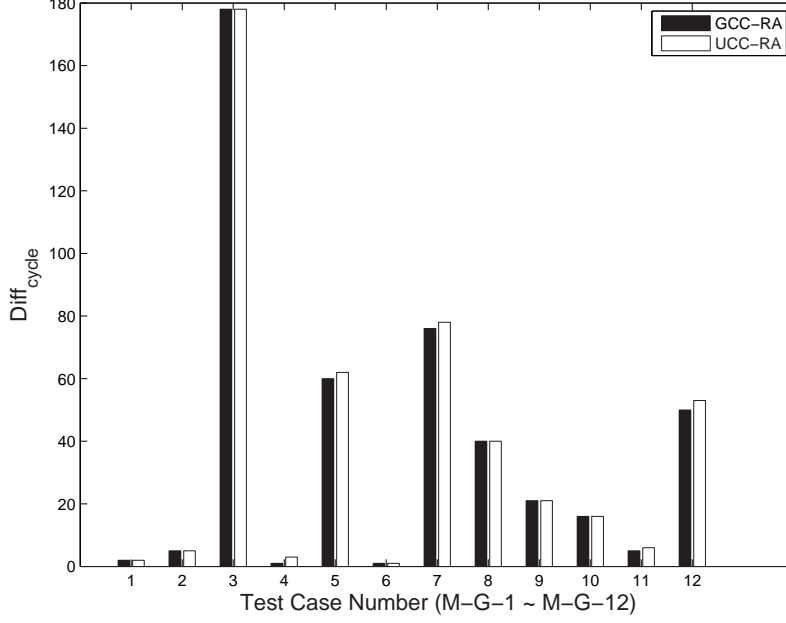
Figure 50: Code quality comparison between UCC-RA and GCC-RA.

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

6.2.1.4 The energy savings

The energy savings per update 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.

$$Diff_{energy} = Diff_{script\_size} \times E_{trans} + Diff_{cycle} \times E_{exe} \times Cnt \tag{6.1}$$

$$EnergySavings = Diff_{energy}^{GCC-RA} - Diff_{energy}^{UCC-RA} \tag{6.2}$$

91

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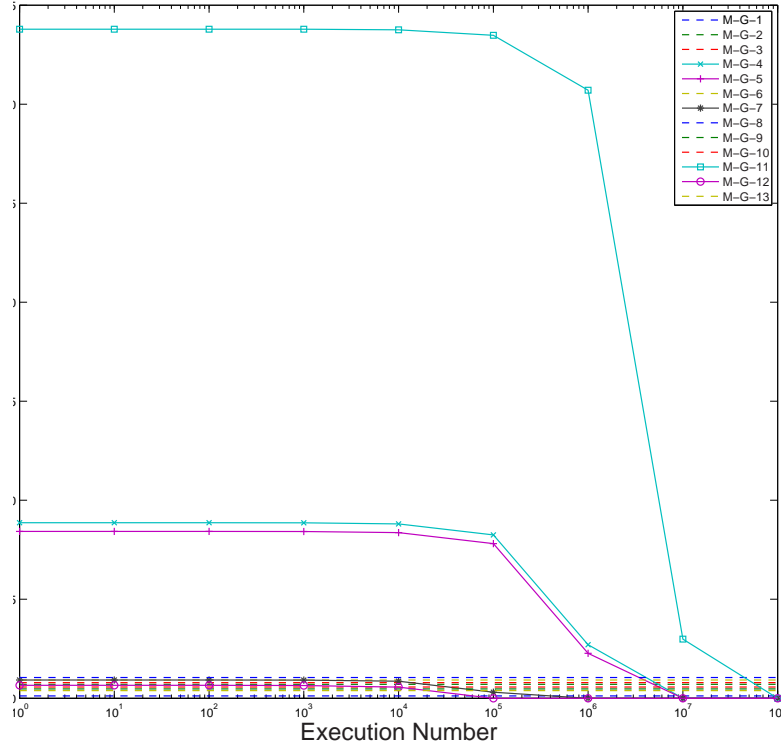

Figure 51: The energy savings per update for general purpose applications.

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.

92

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

6.2.1.5 The problem complexity and compilation time

For a given program, automatically generated general purpose software update benchmarks are used to evaluate the ILP problem complexity which is affected by the number of constrains and the number of decision variables and the compilation time spent solving the ILP problem.

Since the ILP problem is more complex to solve when the number of instructions and variables increase, I discuss the problem complexity in this section. Figure 52 plots the number of constraints as a function of instruction number. I 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 [Berkelaar and *et al.*] requires as a function of (the number of variables × the number of IR instructions) in Figure 53.

An interesting observation I found is that the preferred register tag helps to improve the performance. Comparing to an ILP-based register allocator which allocates register from scratch, the preferred register tag is a hint to the solver and can reduce the number of iterations that solver needs to try. As an extreme case, I also tested misleading preferred register tags, e.g., variables are assigned to the preferred register tag randomly, I found 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 Intel Xeon 3.6GHz processor running Fedora Linux 2.4.21 kernel. The physical memory size is 2GB while in the experiments, the largest observed memory usage is less than 256 MB. Figure 54 shows that the average time required to solve one iteration increase about linearly with the problem complexity. It usually takes the solver less than 175 seconds to allocate registers for a chunk of 250 IR instructions. As a comparison, it takes GCC-RA less than one second to solve the same problem. While UCC-RA is much slower than GCC-RA, it is
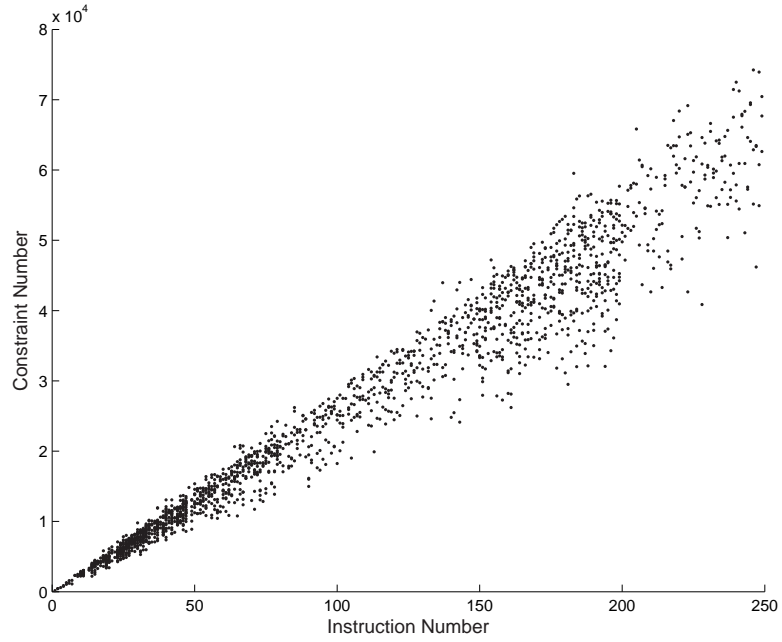
Figure 52: The number of constraints as a function of number of IR instruction.
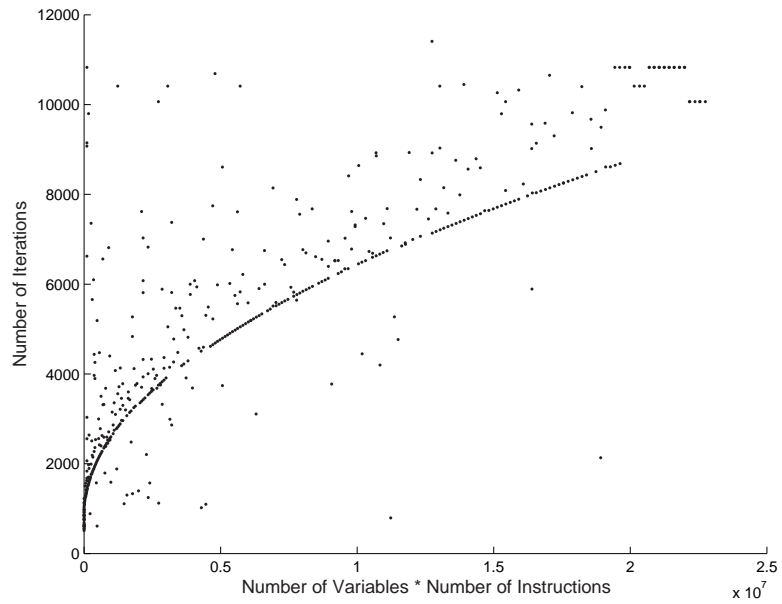

Figure 53: The number of iterations as a function of (the number of variables $\times$ the number of IR instructions).

not a significant problem for WSNs due to the following reasons: (i) sensor applications are small programs limited by the memory size of the sensor node; (ii) UCC-RA is applied only to the identified *changed* chunks instead of the complete functions or the whole application; (iii) it is worthwhile to trade the compilation time at the server side, where both energy and computation power are abundant, for the energy savings on sensor nodes where resources are highly constrained.
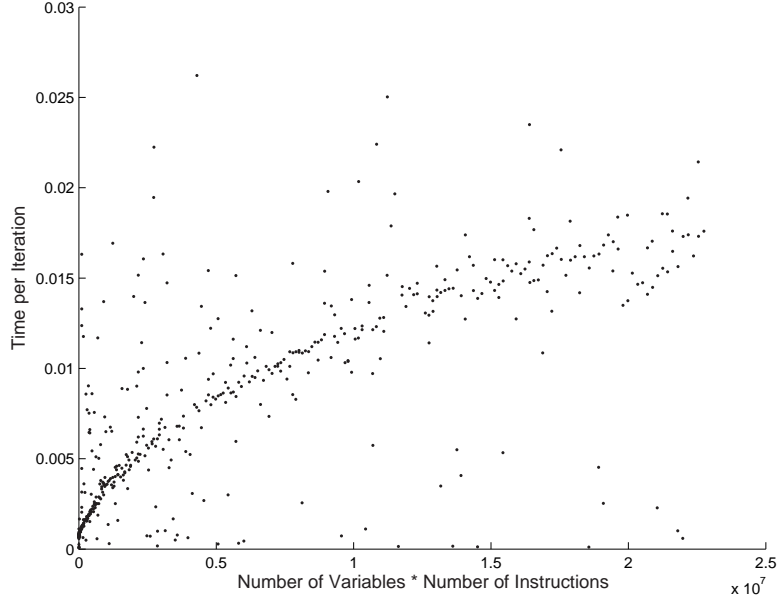

Figure 54: The time to solve one iteration as a function of (the number of variables × the number of IR instructions).

I also performed experiments on testing 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 that solving non-linear problems is orders of magnitude slower than a linear problem.

6.2.2   General purpose software update 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 manual generated general purpose

benchmarks (M-G-14 $\sim$ M-G-19) list in Figure 47 to generate the binary images and further the patch scripts.

The tradeoff of UCC-DA is between the stack size and the generate script size. Keeping more local variables in the same location as the previous version reduces the update script size. However, it may increase the stack size because when variables are removed, the memory holes will not be filled because we want to keep the data allocation similarity. Thus, in the experiments, I evaluated both the generated script size and the worst case stack usage using different data allocation algorithms.

6.2.2.1   Settings

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 to get the baseline binary.

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.The wasted space threshold $SpaceT$ is set to 5 Bytes for the experiments.

Then, I used `tos-ramsize` [50], the tool included in TinyOS release, to generate the worst case stack size of the binary generated using different data allocation schemes.

I used the TinyOS applications as my benchmarks. The simple applications such as Blink, CntToRfms and CntToLeds, do not use many local variables, because there is very little computation involved. Thus, I chose the Advanced Encryption Standard (AES) application [52] as the benchmark. 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 that 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.

The update benchmarks are created based on the AES application, shown in Figure fbench.chg1 M-G-14 $\sim$ M-G-19. The medium size update includes code update that add or remove multiple local variables in a single function. The large size update includes code update that add or remove multiple local variables in multiple functions.

96

6.2.2.2   The generated script size

I first used UCC-DA and GCC-DA to generate the new binary, compared the new binary with old one to generate the update script. The script size comparison is shown in Figure refda-upd. I set the wasted space threshold $SpaceT$ to be 5 Bytes for the experiments.
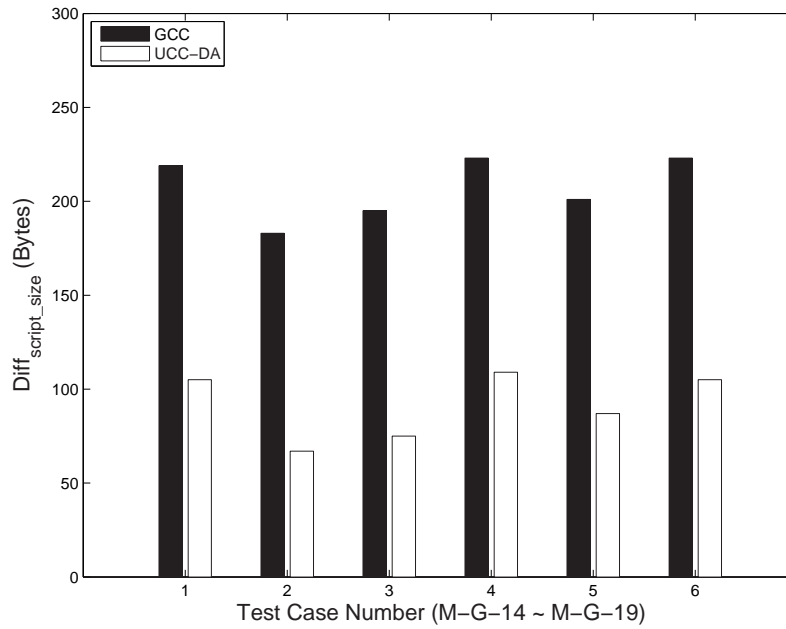

Figure 55: Script size comparison between UCC-DA and GCC-DA.

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 reallocated. For the deleted variables, the memory hole will be left unfilled, if the total wasted memory size is smaller than the threshold $Space_T$. Otherwise, the variable are selected to the fill up the holes based on the factor that is presented in 3.4. Under the given threshold, 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.

6.2.2.3 The wasted memory space

The UCC-DA algorithm trades the memory space for the script size reduction. Keeping the unchanged variables in place may cause memory holes if some variables are removed. Thus, it may cause some memory waste and result in a larger worst-case stack size. Figure 56 compares the worst-case stack size of the binaries generated by different data allocation algorithms.
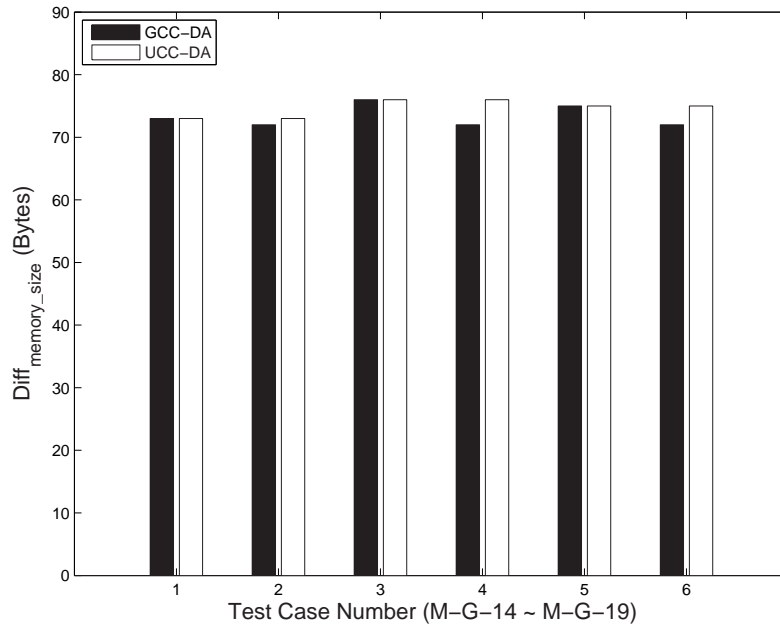

Figure 56: Worst-case stack size comparison between UCC-DA and GCC-DA.

From the experiment results, using UCC-DA only increases the memory usage by 1.9% on average, yet reduces the script size by 56%. This is because only when the memory space taken by the removed variables is larger than the memory space needed by the inserted variables, the memory space may be wasted. In real life, the most common reasons of code updates are adding new features or fixing existing bugs. Changing existing code and adding new code are more likely to happen instead of removing existing code.

6.2.2.4 Tradeoff between wasted space and binary differences

Shown in Figure 55, having the wasted space threshold *SpaceT* set to be 5 Bytes gives 56% script size deduction. Reducing this threshold may increase the script size, because more variables will need to be moved to fill up the memory holes in order to meet this restriction.

Figure 57 shows such tradeoff. I counted the number of instructions that need to be updated due to data reallocation using GCC-DA. Then, I compared that with the number of instructions that need to be updated due to data allocation using UCC-DA, to compute the saved update %. With high wasted spaces threshold, the saved update % is higher. Vice versa.
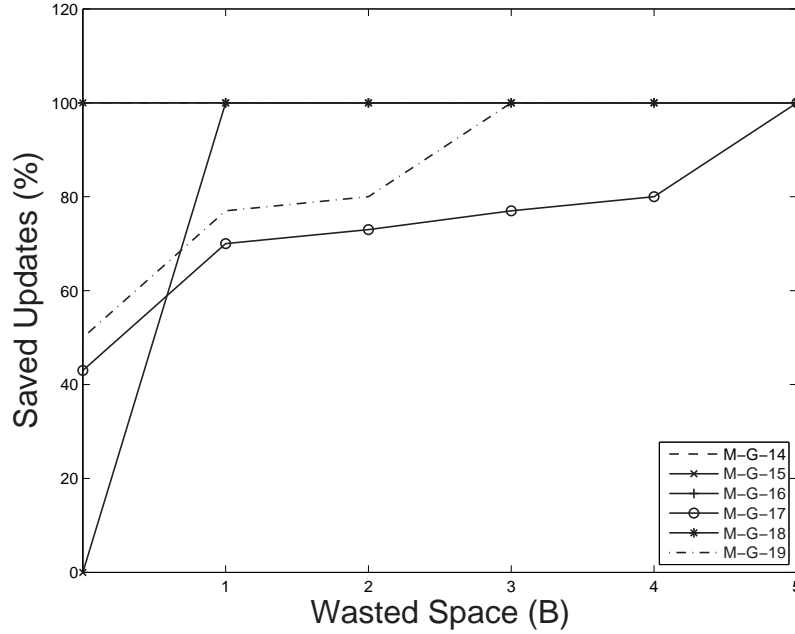

Figure 57: Tradeoff between the worst-case stack size and the instruction updates.

From figure 57, we can also see that the wasted space does not have to be high to tolerant the instruction updates caused by data reallocation. The worst-case stack size of the AES application is 67 Bytes. A 5 Byte memory waste is large enough to tolerant all instruction updates caused by data reallocation, in the given test cases.

An interesting observation is that even setting the threshold *SpaceT* to be 0 Bytes can give some improvement, compared to the GCC-DA scheme. For example, test case M-G-14, M-G-16, and M-G-18 achieve 100% update deduction even when the threshold is set to 0

Byte. This is 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 always allocated on top to the unchanged ones. Thus, these unchanged variables will not be reallocated, so that less instruction updates will be caused.

6.2.3   General purpose software update using the integrated scheme

6.2.3.1   Performance evaluation using man-benches

The experimental results in 6.2.1 and 6.2.2 show that using the 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%.

Integrating both the UCC-RA and UCC-DA schemes should combine the benefits caused by both algorithms and reduce the script size even more. I implemented the integration algorithm proposed in 3.1.3 and ran the integration algorithm on the manual cases 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 58.

As shown in Figure 58, the integrated scheme produce 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 run-time overhead. The reason is that the sensor applications are very simple that the it has a low register pressure. For the presented test cases, the UCC-RA scheme can always find free registers to store the variables that are not tagged with a preferred register. Thus, no extra move or spill instructions are introduced.
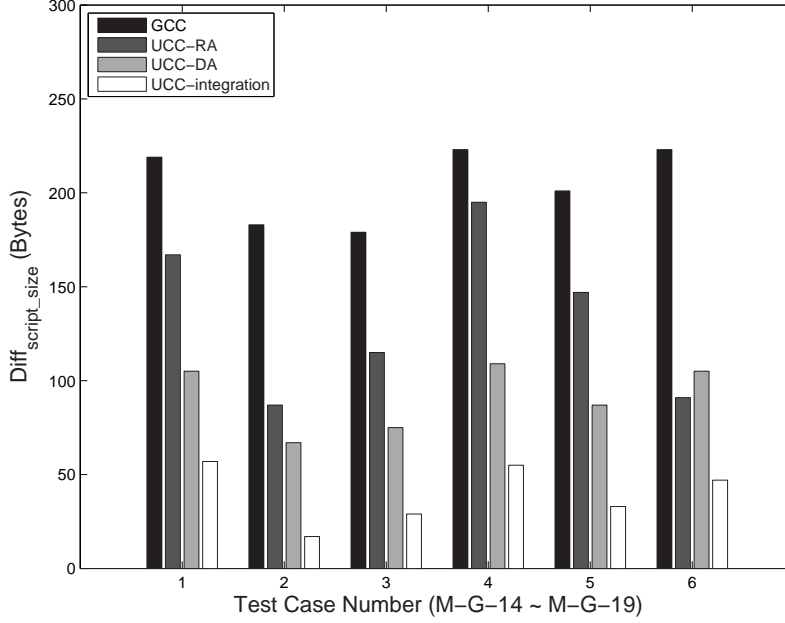
Figure 58: Script size comparison between the integrated scheme and the baseline scheme.

6.2.3.2 Performance evaluation using real-benches

I used the real general purpose benchmarks (R-G-1 $\sim$ 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 according using update primitives described before. These scripts are distributed to the network using the proposed MCP protocol and Deluge protocol to compare the network traffic and time savings.

Figure 59 shows the comparison results which include the number of script instructions per each type of primitive, and the final script size (bytes) for these two compilation techniques. In the case study, I did not add new instructions such that the performance is the same.

The average script size deduction for all the 6 test cases is 55% comparing with GCC. This is because UCC reduces the instructions that need to be updated, thus the number of update primitives in the script is reduced. In addition, I found that when more instructions

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

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 `replace` primitive is more efficient. For example, if there are `N` instructions that can be "cloned" from the original code, which means that they are the same with the related instructions in the original code except for the register assignments, and a one-one mapping can be created between the register assignments in the two versions. The number of such register assignment mappings is `M`. In such situation, I can use both `clone` primitive and `replace` primitive to represent the code updates. And the overhead of using each primitive is shown in the following equations.

$$Cost_{clone} = 5 + 2 * M \tag{6.3}$$
$$Cost_{replace} = 1 + 2 * N \tag{6.4}$$
$$Cost_{clone} < Cost_{replace} \Rightarrow N - M > 2 \tag{6.5}$$

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

6.2.4 DSP software update pre-dissemination

In order to compare the performance of the proposed update-conscious compilation data allocation scheme (GCC-DA) for DSP applications and the CSOA/CGOA schemes, I used the manual generated DSP benchmarks (M-D-1 $\sim$ M-D-5) list in Figure 45 and Figure 48 to generate the binary image and further the patch script using the proposed script primitives. Then, I used the automatically generated DSP benchmarks to study the performance tradeoffs for more general cases.

6.2.4.1 Settings

I have implemented the proposed update-conscious ICSOA/ICGOA algorithms. I chose the Lance Compiler[2] to convert the source code (C code) into intermediate representations (IRs) from which the access sequence and interference graph are extracted. I selected the DSPstone[1] benchmark suite that is widely used to measure the performance of DSP compilers. I adopted CSOA-Offsetstone[3] as the baseline CSOA and implemented ICSOA on top of it. new interferences conflict with existing variable partitioning result. An interference conflict happens when two coalesced variables (in the old assignment) have overlapped live ranges and thus cannot be coalesced anymore.

6.2.4.2 Performance evaluation using man-benches

Single offset assignment Figure 60 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 functional primitives;
- *Advanced code update script* that uses all types of the functional primitives;
- *Context-aware update script* that uses both functional and data primitives.

Using the same script generator with ICSOA, the update script size can be reduced by 32%. This is because that the update-aware scheme 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 contains removal or update in the code update, the two script generators produce the same script i.e. Test-Case M-D-4 and M-D-5.

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-Case M-D-4 and M-D-5. I see a significant script size reduction by using this scheme. Adopting context-aware script tends to incur large complexity i.e. Test-Case M-D-1 and M-D-3 where I see a small script size increase due to the complexity to specify the offset assignment change.
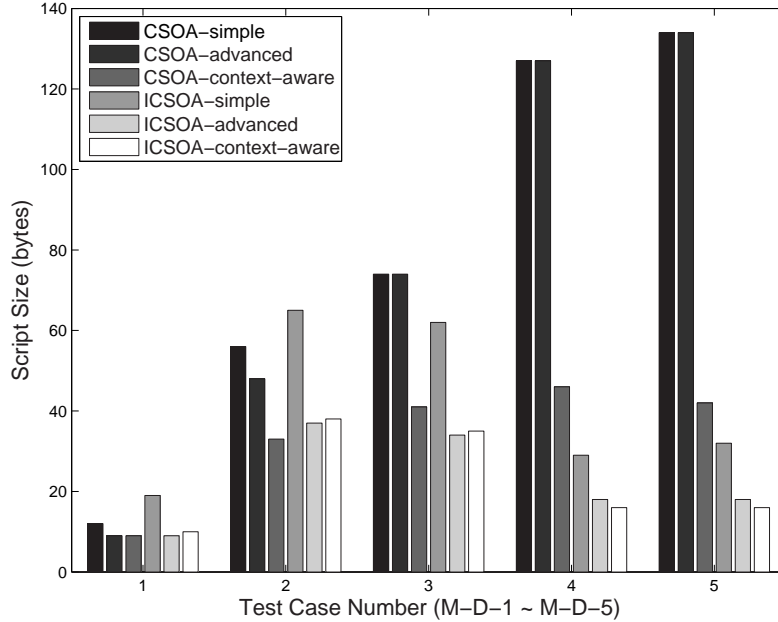

Figure 60: Script size comparison between ICSOA and CSOA ($Num_{addr\_reg} = 1$).

General offset assignment When there are multiple ARs, Figure 61 compares CGOA and ICGOA schemes with the different script generators.
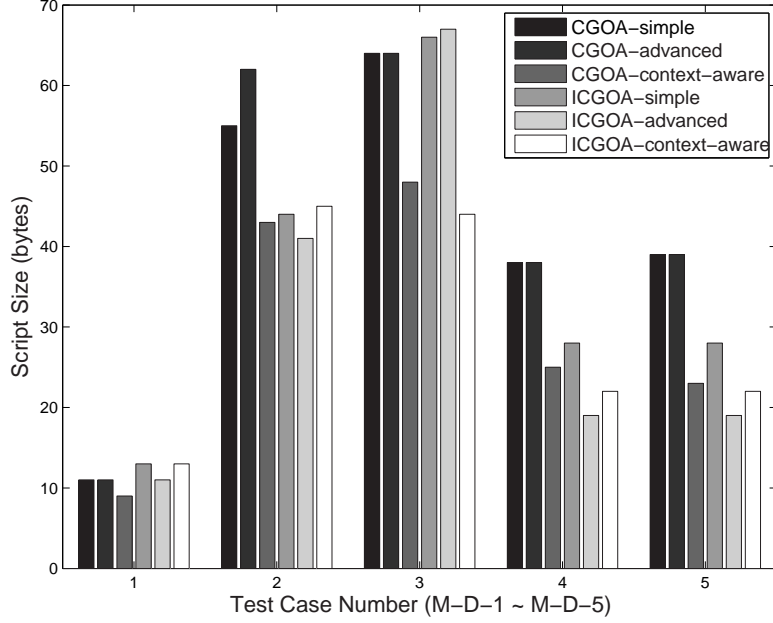
Figure 61: Script size comparison ICGOA and CGOA($Num_{addr\_reg} = 2$).

When there are more ARs, recompiling the program results in large changes in both the variable partition and offset assignment. For Test-Case 3, CGOA with context-aware script has larger size than that with simple script. This is because that the significant variable partition change and requires more primitives to specify the new offset layout.

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

In this paper I evaluated the static code quality i.e. the number of instructions in the new binary produced by CSOA and ICSOA schemes. An alternative approach is to evaluate the dynamic code quality i.e. the runtime instruction counts with given execution profiles. Although the latter provides more accurate evaluation, as I discussed in the introduction section, embedded mobile systems can periodically recharge the battery, so the execution overhead is less critical compared to its the communication overhead.

Single offset assignment As shown in Figure 62, ICSOA produces about the similar number of instructions as CSOA. On average, the binary generated by ICSOA is 10% larger than the binary generated by CSOA. And for the worst test case, i.e. Test-Case 3, the binary

generated by ICSOA is 23% larger than CSOA. Because the ICSOA scheme incrementally does the data allocation based on the *coalesced access graph* of the old version, the old variable coalescing result is kept in the new version to improve code similarity. As a result, the code generated by ICSOA is not as efficient.
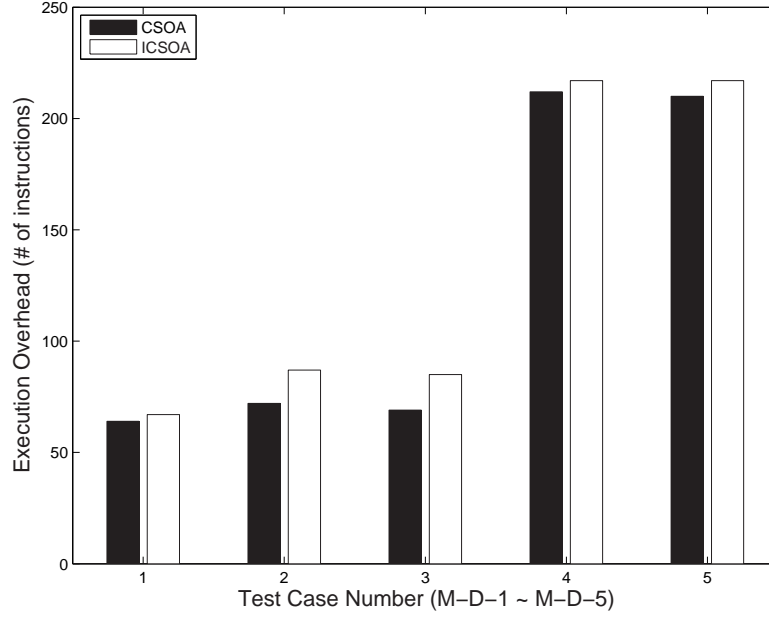

Figure 62: Code quality comparison between CSOA and ICSOA.

To better understand the code quality difference between two approaches, Figure 63 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 create their mapping to the binary level code segments.

Test	CSOA				ICSOA			
Case#	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 63: Execution overhead breakdown.

106

Due to the change to offset assignment, the same IR instructions may be different in the old and new code. The change could be categorized into 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 update does not change the instruction number as no extra instruction is added, but for the second type, one extra instruction is added per change.

To study the code quality, I divide the overhead into four categories as follows. T1-T3 shows 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 63, I found that CSOA generates less binary instructions for the unchanged IR part. It removes more AR loading instructions, and inserts less such instructions. For the new code part, CSOA generates less AR loading instructions. When performing complete recompilation, CSOA uses the new access sequences and variable interferences of the whole function, and thus can generate the 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 "move" instructions in the code, which causes 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 64. 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
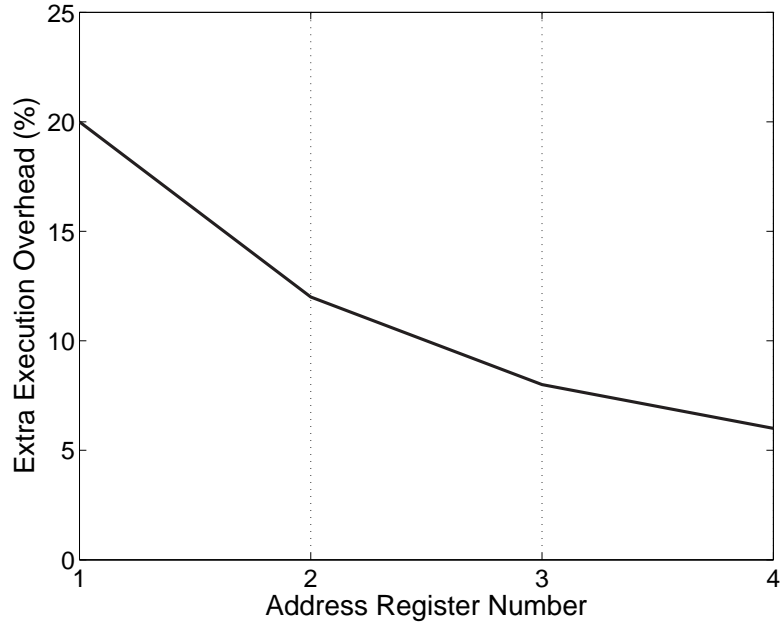
Figure 64: Code quality comparison between ICGOA and CGOA.
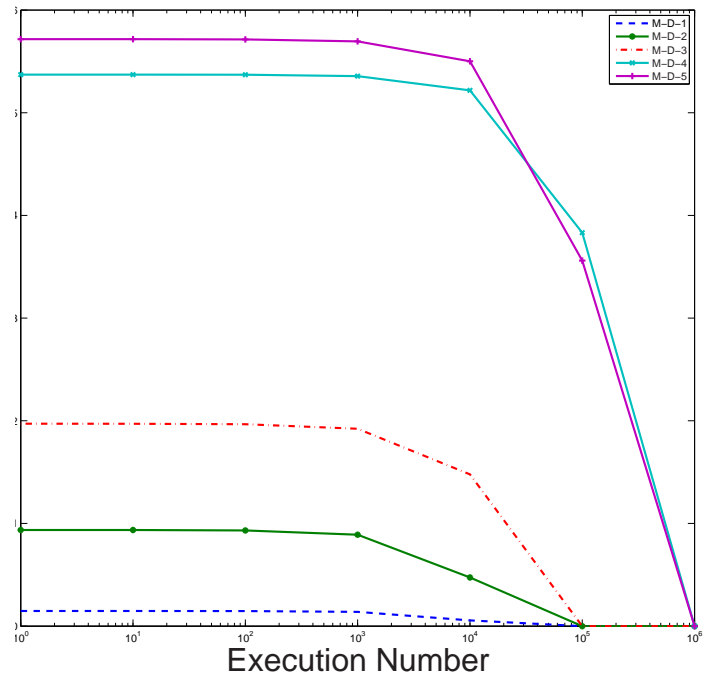

Figure 65: The energy savings for DSP applications.

result in less overhead in ICSOA.

The update-conscious data allocation scheme trades the run-time code performance for the transmission overhead during software update. Thus, the overall energy savings depend on the number of the times that the target binary will execute before retiring. Figure 65 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.

As we can see in Figure 65, with the increase of the execution number of one application, the overall energy savings that are achieved by using the update-conscious compilation scheme is 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.

6.2.4.3 Performance evaluation using auto-benches

I next 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 66. For all three types of the script generation schemes, incremental compilation scheme reduces more of the update script size and thus the software update transmission overhead. However, the results show that I 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.

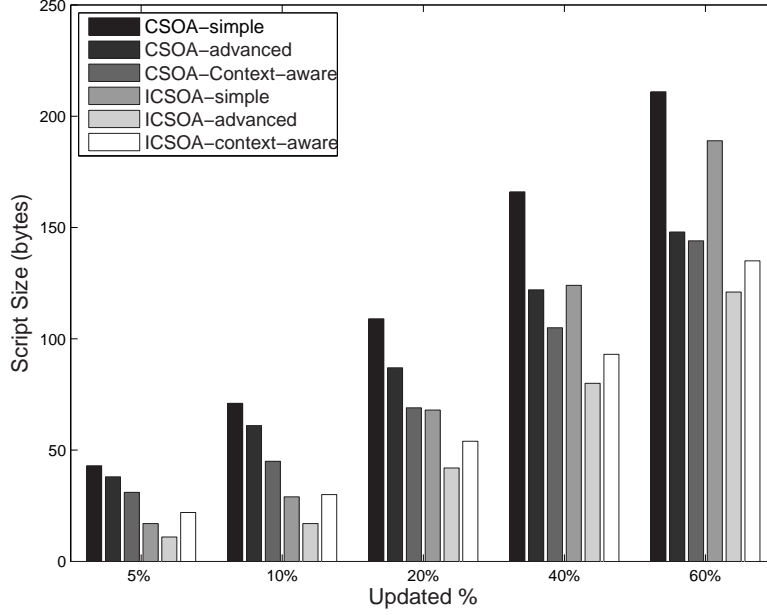The code quality is compared in Figure 67. Larger code update percentage, i.e. over

Figure 66: Script size comparison (scattered random new code insertion).

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

From Figure 67 and Figure 66, I conclude that when the code update percentage is between 20% and 40%, using the update-conscious offset assignment scheme can save about 30% of the transmission overhead, assuming that advanced script is used, with about 2% extra instruction execution.

From the experimental results, I can also see that the simply using the code primitives works better with the incremental compilation scheme, and the context-aware primitives works better with the recompilation scheme. This is because that the context-aware primitives trades updating individual instructions for setting up the auxiliary data structures and letting the sensors to construct these updates. Thus, when I recompile the new version, a relatively great number of instructions are changed due to the data allocation differences, so the context-aware primitives can gain more benefit by saving those updates. On the other
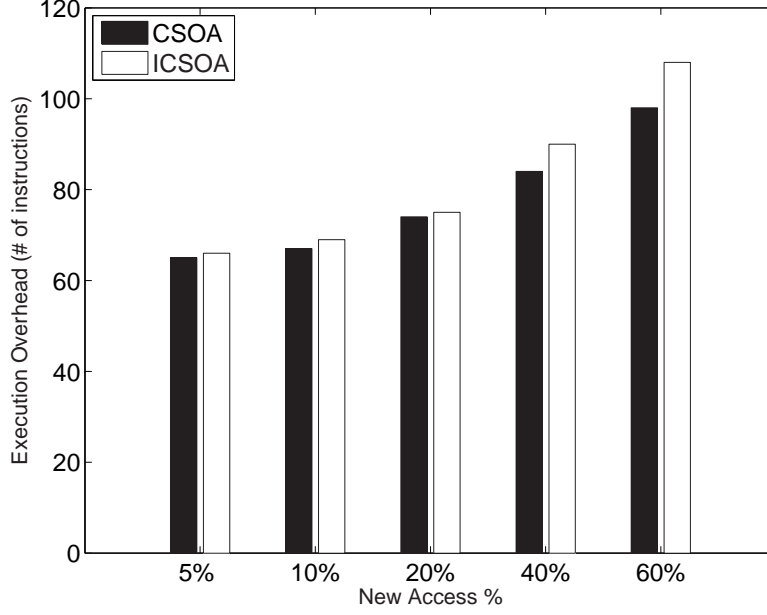
Figure 67: Code quality comparison (scattered random new code insertion).

hand, when I use the incremental compilation technique, the saving is not great enough to balance the spending in setting up the data structures, therefore, just using the code primitives is more beneficial.

6.2.4.4 Performance evaluation using real-benches

I used the real general DSP benchmarks (R-D-1 $\sim$ R-D-3) listed in Figure 45 to study the performance tradeoffs and energy savings for real software update cases.

Considering only the update-conscious data allocation scheme (UCC-DA) for DSP applications, I compared the update script size, generated binary performance and long term energy savings between the baseline CSOA scheme and the proposed ICSOA scheme. With the consideration of both update-conscious data allocation (UCC-DA) and update-conscious register allocation (UCC-RA), I did the comparison between the baseline CGOA scheme and ICGOA scheme. The update scripts are then generated using different update primitives described before. These scripts are distributed to the network using the base Deluge protocol and the proposed MCP protocol to compare the network traffic and time consumption.
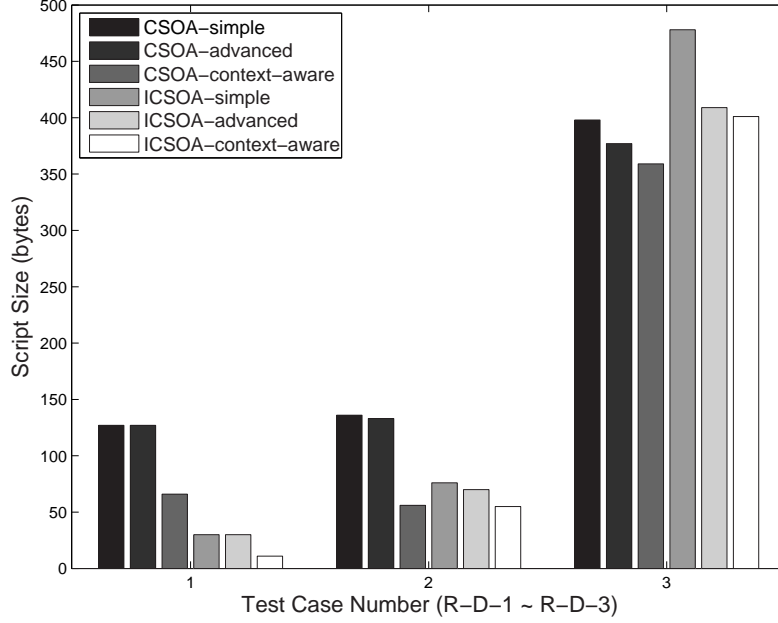
Figure 68: Script size comparison between CSOA and ICSOA ($Num_{addr\_reg} = 1$).

Figure 68 shows the generated script size comparison using different compilation techniques and update primitive sets. Here, test case R-D-1 and R-D-2 are medium level updates, while R-D-3 is a high level update.

Comparison between **CSOA-simple** and **ICSOA-simple**, we can see that update-conscious data allocation (ICSOA) produces the binary that is more similar as 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 changes, the old and new code segments are mixed, such that the benefit from keeping the old data offset assignment diminishes.

Comparison between **CSOA-simple** and **CSOA-advanced** shows the script size deduction that we can achieve using the advanced script primitives over the simple script primitives. 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 inlined functions, the `clone` primitive can help to reduce the script size, and higher level updates provide more opportunities for these primitives to take effect.

Comparison between **CSOA-simple** and **CSOA-data** shows the script size deduction

that we can achieve using the context-aware primitives over the simple script primitives. Although using the context-aware primitives primitives increase the code regeneration overhead on the sensors, it helps to reduce the script size significantly, especially when the data allocation update is significant. For example, the baseline CSOA scheme compared to the ICSOA scheme produces a more different data allocation result from the old version. The improvement achieved by the context-aware primitives when using the CSOA compilation technique is higher than using the ICSOA compilation technique.

In real life, there are usually more than one address registers available on a DSP chip. The more the address registers that are available, the larger the solution space is. Without the knowledge of the old compilation results, regenerating the new binary may cause more data allocation differences.

Figure 69 shows the experimental results, for double address register situation. Here, I compared the baseline CGOA scheme with the proposed ICGOA scheme.
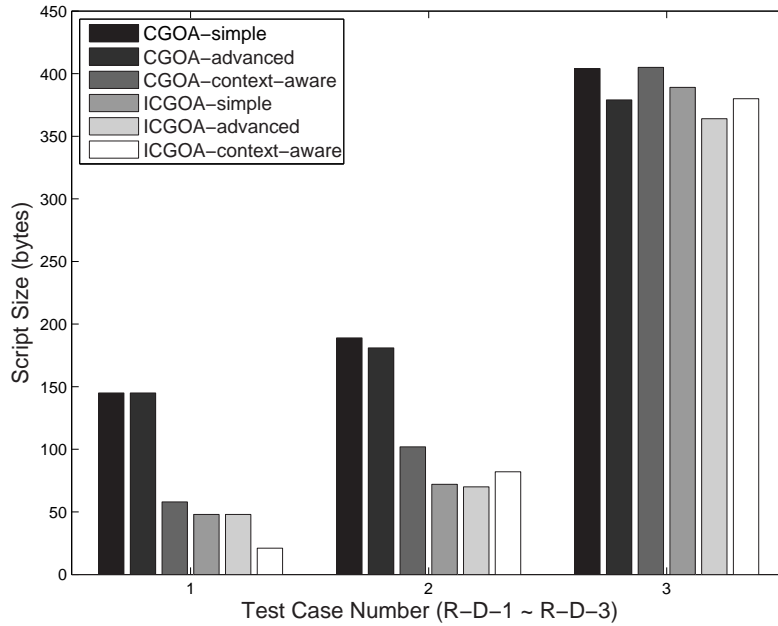

Figure 69: Script size comparison between CGOA and ICGOA ($Num_{addr\_reg} = 2$).

We can see that when more address registers are available, the baseline scheme tends to generate the larger update scripts, which increases the improvement space of the update-conscious compilation techniques.

Figure 70 shows the generated code performance comparison between the baseline CSOA scheme and the proposed ICSOA scheme. When code changes, inheriting the old data allocation result may not result in an 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%.
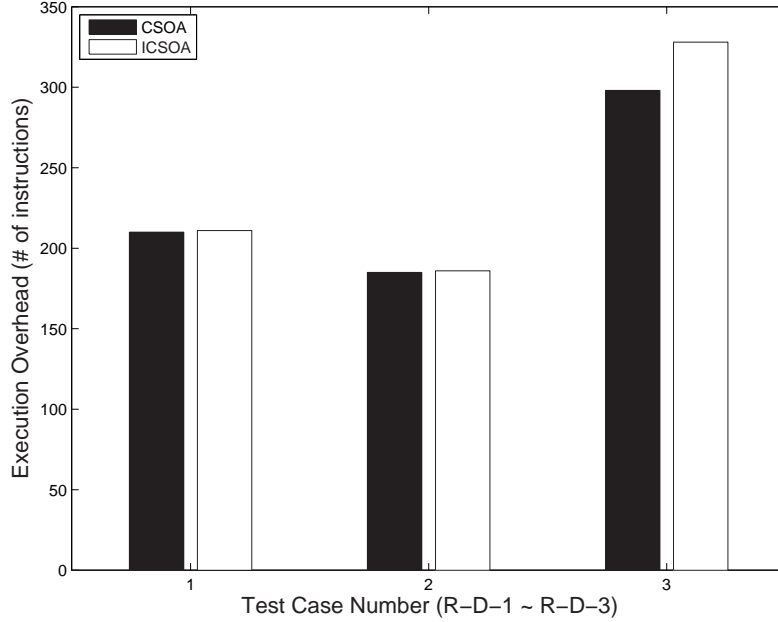

Figure 70: Code quality comparison between CSOA and ICSOA ($Num_{addr\_reg} = 1$).

6.3   PATCH DISSEMINATION PERFORMANCE EVALUATION

6.3.1   Settings

I implemented MCP on the TinyOS [5] platform. For comparison, I also implemented Melete [61] and studied various network settings using TOSSIM [4].

I simulated mesh MA-WSNs of different sizes. I set the default spacing factor to 15 and model 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%

114

of the other sensors to run `A`. MCP disseminates the code from in-network sources instead of the sink.
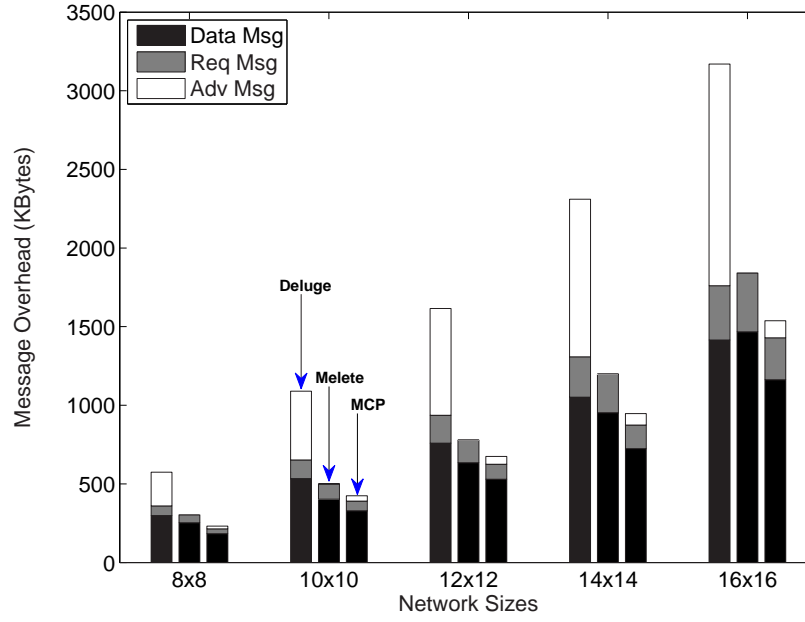
6.3.2   Message overhead

Figure 71: Message overhead.

Fig. 71 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 [61]. There are a large number of `ADV` messages in Deluge, and a negligible number in Melete. The reason of such difference is Deluge depends heavily on incoming `ADV` message, e.g., a sensor node only sends out new requests if it receives `ADV` messages indicating its neighbors have more up-to-date data. Instead, 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 receiving one complete page. The `ADV` messages contribute to 37-40% of the total message overhead in Deluge.

My scheme takes a similar approach as 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 about 20% of the message overhead from Melete. 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 chosing one closeby source, which reduces the number of data packets in transmission.
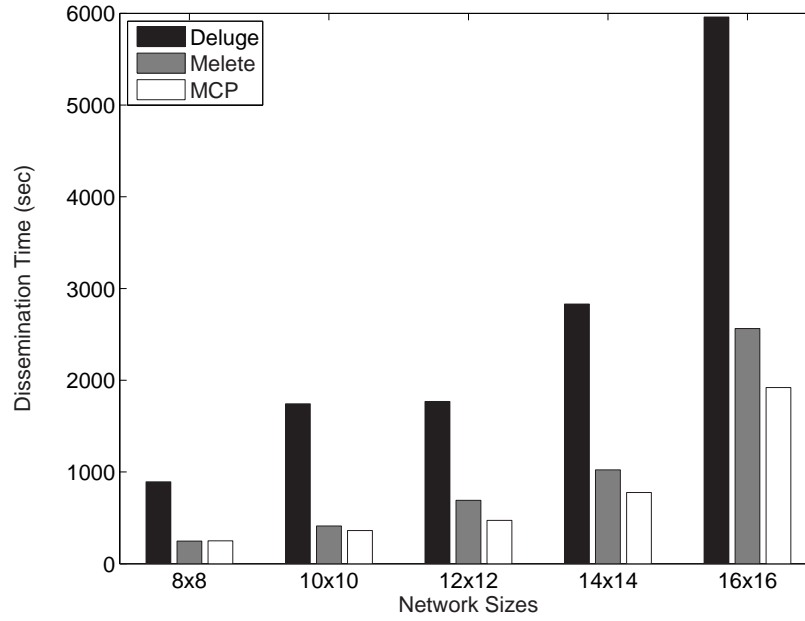
6.3.3 Completion time

Figure 72: Dissemination time.

Fig. 72 compares the dissemination completion time of the different protocols. For the Deluge result, I record the time interval used by all requesters to complete the new code downloading. 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.
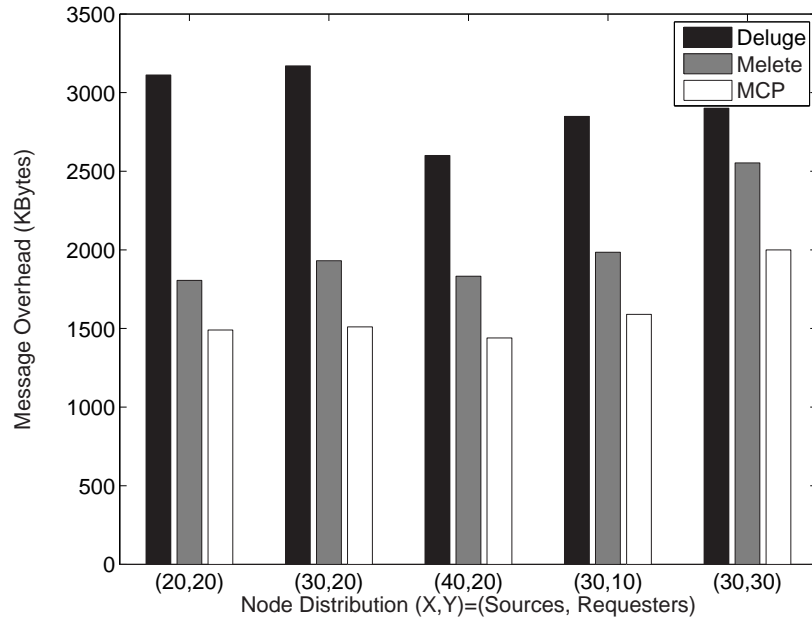
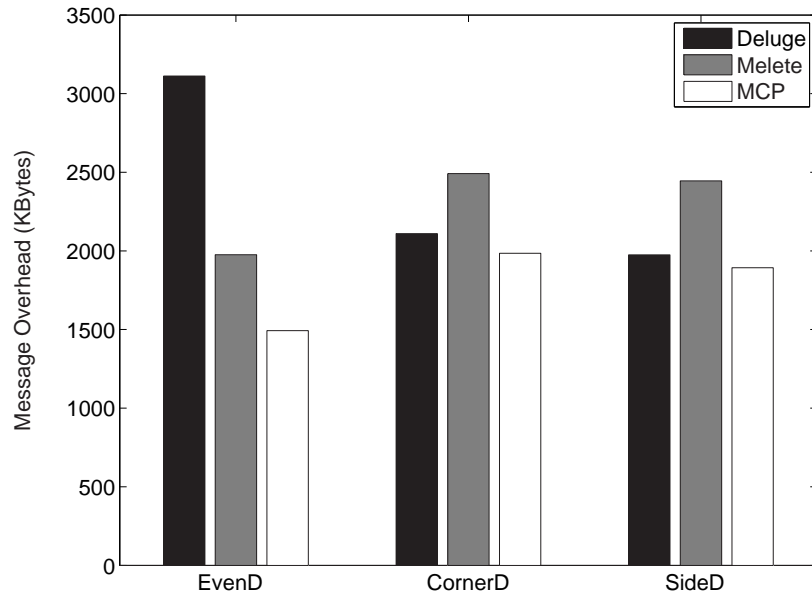Figure 73: Dissemination with different number of sources and requesters.

Figure 74: Dissemination with uneven source/requester node distribution.

6.3.4 Sensitivity to node distribution

Fig. 73 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.

Fig. 74 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 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.

6.3.5 Sensitivity to application sizes

Fig. 75 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.

6.3.6 Sensitivity to cache sizes

Fig 76 summarizes 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 there is no caching. From the figure, MCP achieves significant communication overhead reduction when
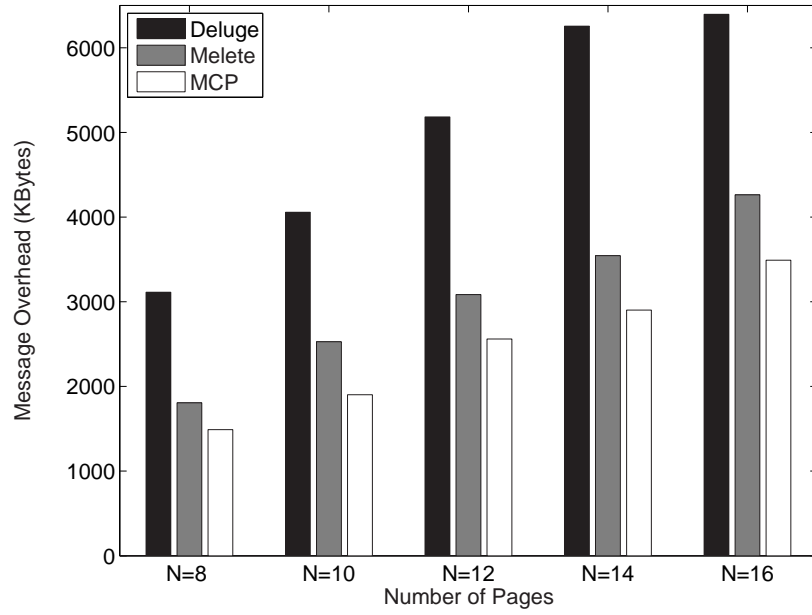
Figure 75: Dissemination with different number of pages.
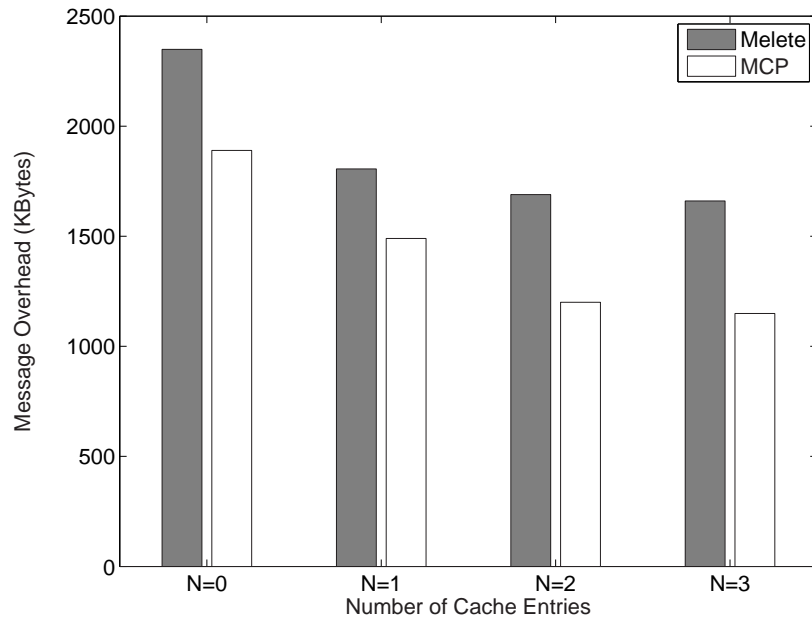
Figure 76: Dissemination with Different Cache Sizes.

caching one or two future pages, and diminishing benefits when 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 bigger 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

In this chapter, I will address the future research directions and the conclusion of this research.

7.1   FUTURE WORK

Although the designed software update framework has achieved the design goal and gained a lot energy savings for the WSN platform, there are some directions that I can keep working on to increase its value.

7.1.1   Apply to different platforms

The software update management framework can be used in the hardware platforms other than WSNs, e.g. the smart phones. The popularity of the smart phones has brought more interest of developing and using applications on them. By September 1st, 2010, there have been over 250,000 applications launched on the iPhone [19] platform. According to Tech Crunchies [25], the average number of applications installed on an iPhone is 65. Because of the rapid growing demand and the fast developing pace, these applications tend to be updated very often. Multiple releases of one application could be launched in a month. How to efficiently update these applications could be an issue, because frequent software update may use up the energy stored in the battery and consume too much bandwidth to satisfy the QoS of the other running applications.

Applying the update-conscious compilation and the 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 multi-cast based code dissemination protocol installed, the smart phones will be able to download the new applications from the peer phones via bluetooth, which will reduce the data usage and cause less affect to the other running applications that may be heavily using the network.

One future research direction is to apply the techniques proposed in this research to smart phones and evaluate the benefits that it can gain compared to the traditional solutions.

7.1.2   Approach 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. Besides this, there are other UCC research opportunities, such as UCC instruction selection and UCC instruction scheduling.

Instruction selection transforms the tree-based middle-level intermediate representation (IR) into a low-level IR 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. 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 the run-time overhead but also the code similarity. This trade-off between the run-time overhead and the transmission overhead can be studied.

The functional update primitive design favors the continuous updates, because in that way we can use one primitive to describe multiple updated instructions. Comparing two updates that have the same number of new instructions, where one has all the updates concentrating at one or two update points, and the other one has all the updates scattering in the existing code, the first one 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 the run-time performance by introducing more "stalls". This trade-off needs to be studied in the future research.

7.2 CONCLUSION

Wireless sensor networks (WSNs) have been widely used. Making it easy to maintain becomes an issue, because the sensors are usually unattached after deployed and the software running on them needs to updates for various reasons. Without the physical accesses to the sensors, the software update procedure can be energy consuming, because it relies on wireless communication. Experimental results have shown that the energy spent in transmitting one bit one hope away is equivalent to the energy consumed by executing one thousand instructions. If the sensor is running a simple application with a long idle time, the energy spent on the software update procedure may be higher than the energy spent in sensing and reporting.

The greatest benefit that WSNs bring is that, once it is set up, it can sensor, compute and report automatically with very less human effort involved. However, if the energy consumption in the software update procedure is too high, the WSN users may need to frequently change the batteries of the sensors, or redeploys new sensors whenever the old ones run out of power. This obviously makes using WSNs less appealing and of course will hinder the widespread of using this technology in real life. Therefore, how to update the software efficiently in WSNs is a critical problem to solve in WSN study.

In my thesis, I designed a framework that tacks this problem down from different viewpoints.

The update-conscious compiler (UCC) first generates the a new binary that is similar as the old version, in order to reduce the updates that need to be made. 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 heuristic based solution to allocate the local variables, under a certain memory usage constraint. The update-conscious data allocation scheme for DSP applications adds the binary similarity consideration to the existing CSOA, CGOA algorithms, which generates the new binary with not only the similar data allocation

result but also the similar addressing modes for the memory access instructions. Because of the trade-offs described above, the number of executions that the new binary is going to make, the memory size of the target platform, and the memory usage of the new binary all affect the performance of the UCC. It achieves the best performance, when the target software is not frequently executed and it has a lower memory usage.

The generated binary is then compared with the old binary to generate the update patch. In order to furthermore reduce the patch size, several sets of patch primitives are designed. The `simple` functional update primitives directly stores the binary comparison results in the patch, which may result in a larger update patch but is easy to be decoded on the sensors. The `advanced` functional update primitives summarize the updates that affect more than one instructions in one primitive, which can generate smaller patches but requires a more complicated decoder on the sensors. The data based primitives are used to describe the data allocation changes and the affected addressing mode changes. The binary generation on the sensors involves rebuilding the data allocation table and fixing the addressing mode updates. This scheme requires the most effort to decode the patch and reconstruct the new binary compared to the functional update primitive sets. Therefore, it depends on the update level and the network setting to choose the right primitive set. Higher level updates desire more complicated primitives to reduce the patch size.

A multi-case based stateful code distribution protocol (MCP) is then used to disseminate the generated binary to the sensors. This protocol stores the routing information to the sources on the sensors, so that the download request can be directed to the sources without flooding the network. The memory on the sensors is wisely divided to cache packages received from different sources.

The three components together solve a critical WSN problem. According the experimental results, each component contributes a significantly amount of energy saving in the software update procedure. The motivation of the update-conscious compilation techniques is another great contribution of this research.

BIBLIOGRAPHY

[1] DSPStone benchmark suite. Website, 1995. http://iss.rwth-aachen.de/Projekte/Tools/DSPSTONE.

[2] Lance compiler, 2001. http://www.lancecompiler.com/.

[3] Offsetstone benchmark suite, 2003. http://www.address-code-optimization.org.

[4] Tossim, 2003. http://www.eecs.berkeley.edu/~pal/research/tossim.html.

[5] Tinyos, 2004. http://www.tinyos.net.

[6] Atmel atmega128L microcontroller, 2008. http://www.atmel.com/dyn/resources/prod_documents/doc2

[7] 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. ISBN 0-321-48681-1.

[8] A. W. Appel and L. George. Optimal spilling for cisc machines with few registers. In *PLDI' 00: Proceedings of the ACM SIGPLAN 2001 conference on programming language design and implementation*, pages 243–253. ACM Press, 2000.

[9] S. Atri, J. Ramanujam, and M. T. Kandemir. Improving offset assignment on embedded processors using transformations. In *HiPC '00: Proceedings of the 7th international conference on high performance computing*, pages 367–374, London, UK, 2000. Springer-Verlag. ISBN 3-540-41429-0.

[10] K. C. Barr and K. Asanović. Energy-aware lossless data compression. *TOCS' 06: ACM Transactions on Computer Systems*, 24(3):250–291, 2006. ISSN 0734-2071.

[11] D. H. Bartley. Optimizing stack frame accesses for processors with restricted addressing modes. *Software – practice and Experience*, 22(2):101–110, 1992. ISSN 0038-0644.

[Berkelaar and *et al.*] M. Berkelaar and *et al.* Lp_solve 5.5.

[12] M. P. Bivens and M. L. Soffa. Incremental register reallocation. *Software – practice and experience*, 20(10):1015–1047, 1990. ISSN 0038-0644.

[13] P. Briggs, K. D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *TOPLAS' 94: ACM transactions on program languages and systems*, 16(3): 428–455, 1994. ISSN 0164-0925.

[14] G. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. Register allocation via coloring. *Computer languages*, 6:47–57, 1981.

[15] Y. Choi and T. Kim. Address assignment combined with scheduling in dsp code generation. In *DAC '02: Proceedings of the 39th conference on design automation*, pages 225–230, New York, NY, USA, 2002. ACM. ISBN 1-58113-461-4.

[16] F. Chow and J. Hennessy. Register allocation by priority-based coloring. In *Proceedings of the SIGPLAN '84 symposium on compiler construction*, volume 19, pages 222–232, Montreal, Canada, 1984. ACM.

[17] M. Chu, H. Haussecker, and F. Zhao. Scalable information-driven sensor querying and routing for ad hoc heterogeneous sensor networks. *International Journal on High Performance Computing Applications*, 16(3):90–110, 2002.

[18] Coin-or. Nonlinear mixed integer programming, 2006. http://projects.coin-or.org/Bonmin.

[19] A. Corporation. iphone user guide, 2010. http://manuals.info.apple.com/en/iphone_user_guide.pdf.

[20] Crossbow. Mica2 data sheet, 2004. http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/

[21] Crossbow. Micaz data sheet, 2004. http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/

[22] Crossbow. Moteiv telos data sheet, 2004. http://www.moteiv.com.

[23] Crossbow. Telosb data sheet, 2004. http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf

[24] CrossBow. Imote2 data sheet, 2009. http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf

[25] T. Crunchies. Tech crunchies – internet statistics and numbers, 2010. http://www.techcrunchies.com.

[26] 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*, pages 15–28, New York, NY, USA, 2006. ACM. URL http://dx.doi.org/http://doi.acm.org/10.1145/1182807.1182810.

[27] P. K. Dutta, J. W. Hui, D. C. Chu, and D. E. Culler. Securing the deluge network programming system. In *IPSN '06: Proceedings of the 5th international conference on information processing in sensor networks*, pages 326–333, New York, NY, USA, 2006. ACM. ISBN 1-59593-334-4.

[28] C. Fu and K. Wilken. A faster optimal register allocator. In *MICRO '35: Proceedings of the 35th annual ACM/IEEE international symposium on microarchitecture*, pages 245–256, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press. ISBN 0-7695-1859-1.

[29] L. George and A. W. Appel. Iterated register coalescing. *TOPLAS' 96: ACM transactions on program languages and systems*, 18(3):300–324, 1996. ISSN 0164-0925.

[GNU] GNU. http://www.gnu.org/software/diffutils/.

[30] D. W. Goodwin and K. D. Wilken. Optimal and near-optimal global register allocations using 0–1 integer programming. *Software–practice and experience*, 26(8):929–965, 1996. ISSN 0038-0644.

[31] W. R. Heinzelman, J. Kulik, and H. Balakrishnan. Adaptive protocols for information dissemination in wireless sensor networks. In *MobiCom '99: Proceedings of the 5th annual ACM/IEEE international conference on mobile computing and networking*, pages 174–185, New York, NY, USA, 1999. ACM. ISBN 1-58113-142-9.

[32] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *SenSys '04: Proceedings of the 2nd international conference on eEmbedded networked sensor systems*, pages 81–94, New York, NY, USA, 2004. ACM. ISBN 1-58113-879-2.

[33] C. Jaramillo, R. Gupta, and M. L. Soffa. Capturing the effects of code improving transformations. In *PACT '98: Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, page 118, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8591-3.

[34] J. Jeong and D. Culler. Incremental network programming for wireless sensors. In *SECON' 04: Proceedings of sensor and ad hoc communications and networks*, pages 25–33, Oct. 2004.

[35] D. R. Koes and S. C. Goldstein. A global progressive register allocator. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on programming language design and implementation*, pages 204–215, New York, NY, USA, 2006. ACM. ISBN 1-59593-320-4.

[36] J. Koshy and R. Pandey. Remote incremental linking for energy-efficient reprogramming of sensor networks. In *Proceeedings of the 2nd European workshop on wireless sensor networks*, pages 354–365, Jan.-2 Feb. 2005.

[37] P. Lanigan, R. Gandhi, and P. Narasimhan. Sluice: secure dissemination of code updates in sensor networks. In *ICDCS' 06: Proceedings of the 26th IEEE international conference on distributed computing Systems*, pages 53–53, 2006.

[38] P. Lapsley, J. Bier, E. A. Lee, and A. Shoham. *DSP Processor fundamentals: architectures and features.* Wiley-IEEE Press, 1996. ISBN 0780334051.

[39] R. Leupers and F. David. A uniform optimization technique for offset assignment problems. In *ISSS '98: Proceedings of the 11th international symposium on System synthesis*, pages 3–8, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8623-5.

[40] R. Leupers and P. Marwedel. Algorithms for address assignment in dsp code generation. In *ICCAD '96: Proceedings of the 1996 IEEE/ACM international conference on computer-aided design*, pages 109–112, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7597-7.

[41] P. Levis and D. Culler. Maté: a tiny virtual machine for sensor networks. In *ASPLOS-X: Proceedings of the 10th international conference on architectural support for programming languages and operating systems*, pages 85–95, New York, NY, USA, 2002. ACM. ISBN 1-58113-574-2.

[42] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *NSDI'04: Proceedings of the 1st conference on symposium on networked systems design and implementation*, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[43] W. Li, Y. Zhang, J. Yang, and J. Zheng. Ucc: update-conscious compilation for energy efficiency in wireless sensor networks. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on programming language design and implementation*, volume 42, pages 383–393, San Diego, CA, USA, 2007. ACM.

[44] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. Storage assignment to decrease code size. *TOPLAS' 96: ACM transactions on programming languages and systems*, 18 (3):235–253, 1996. ISSN 0164-0925.

[45] 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 *EWSN' 06: Proceedings of the 3rd european workshop on wireless sensor networks*, pages 212–227, 2006.

[46] D. Ottoni, G. Ottoni, G. Araujo, and R. Leupers. Offset assignment using simultaneous variable coalescing. *TECS' 06: ACM Transactions on Embedded Computing Systems*, 5 (4):864–883, 2006. ISSN 1539-9087.

[47] R. Panta, I. Khalil, and S. Bagchi. Stream: Low overhead wireless reprogramming for sensor networks. In *INFOCOM' 07: Proceedings of the 26th IEEE international conference on computer communications.*, pages 928–936, May 2007.

[48] M. Poletto and V. Sarkar. Linear scan register allocation. *TOPLAS' 99: ACM transactions on program languages and systems*, 21(5):895–913, 1999. ISSN 0164-0925.

[49] A. Rao and S. Pande. Storage assignment optimizations to generate compact and efficient code on embedded dsps. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999*

conference on Programming language design and implementation, pages 128–138, New York, NY, USA, 1999. ACM. ISBN 1-58113-094-5.

[50] J. Regehr. Tinyos stack analysis, 2009. http://docs.tinyos.net/index.php/Stack_Analysis.

[51] N. Reijers and K. Langendoen. Efficient code distribution in wireless sensor networks. In *WSNA '03: Proceedings of the 2nd ACM international conference on wireless sensor networks and applications*, pages 60–67, New York, NY, USA, 2003. ACM. ISBN 1-58113-764-8.

[52] V. Rijmen and J. Daemen. Advanced encryption standard, 1998. http://en.wikipedia.org/wiki/Advanced_Encryption_Standard.

[53] M. Ros and P. Sutton. A hamming distance based vliw/epic code compression technique. In *CASES '04: Proceedings of the 2004 international conference on compilers, architecture, and synthesis for embedded systems*, pages 132–139, New York, NY, USA, 2004. ACM. ISBN 1-58113-890-3.

[54] M. Ros and P. Sutton. A post-compilation register reassignment technique for improving hamming distance code compression. In *CASES '05: Proceedings of the 2005 international conference on compilers, architectures and synthesis for embedded systems*, pages 97–104, New York, NY, USA, 2005. ACM. ISBN 1-59593-149-X.

[55] 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.

[56] A. Sudarsanam, S. Liao, and S. Devadas. Analysis and evaluation of address arithmetic capabilities in custom dsp architectures. In *DAC '97: Proceedings of the 34th annual conference on Design automation*, pages 287–292, New York, NY, USA, 1997. ACM. ISBN 0-89791-920-3.

[57] B. Titzer, D. Lee, and J. Palsberg. Avrora: scalable sensor network simulation with precise timing. In *IPSN '05: Proceedings of the 4th information processing in sensor networks, 2005.*, pages 477–482, April 2005.

[58] O. Traub, G. Holloway, and M. D. Smith. Quality and speed in linear-scan register allocation. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 142–151, New York, NY, USA, 1998. ACM. ISBN 0-89791-987-4.

[59] C. von Platen and J. Eker. Feedback linking: optimizing object code layout for updates. *LCTES' 06: Proceedings of conference on languages, compilers, and tools for embedded systems*, 41(7):2–11, 2006. ISSN 0362-1340.

[60] L. Wang. Mnp: multihop network reprogramming service for sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on embedded networked sensor systems*, pages 285–286, New York, NY, USA, 2004. ACM. ISBN 1-58113-879-2.

[61] Y. Yu, L. J. Rittle, V. Bhandari, and J. B. LeBrun. Supporting concurrent applications in wireless sensor networks. In *SenSys '06: Proceedings of the 4th international conference on embedded networked sensor systems*, pages 139–152, New York, NY, USA, 2006. ACM. ISBN 1-59593-343-3.

[62] X. Zhuang, C. Lau, and S. Pande. Storage assignment optimizations through variable coalescence for embedded processors. *LCTES' 03: Proceedings of the 2003 ACM SIGPLAN conference on language, compiler, and tool support for embedded systems*, 38(7): 220–231, 2003. ISSN 0362-1340.

[63] X. Zhuang and S. Pande. An optimization framework for embedded processors with auto-addressing mode. *TOPLAS' 10: ACM Transactions on Programming Languages and Systems*, 32(11), 2010.