An Efficient Code Update Scheme for DSP Applications in Mobile Embedded Systems

Weijia Li , Youtao Zhang
Computer Science Department,University of Pittsburgh,Pittsburgh, PA 15260
{weijiali,zhangyt}@cs.pitt.edu

Abstract

DSP processors usually provide dedicated address generation units (AGUs) to assist address computation. By carefully allocating variables in the memory, DSP compilers take advantage of AGUs and generate efficient code with compact size and improved performance. However, DSP applications running on mobile embedded systems often need to be updated after their initial releases. Studies showed that small changes at the source code level may significantly change the variable layout in the memory and thus the binary code, which causes large energy overheads to mobile embedded systems that patch through wireless or satellite communication, and often pecuniary burden to the users.

In this paper, we propose an update-conscious code update scheme to effectively reduce patch size. It first performs incremental offset assignment based on a recent variable coalescing heuristic, and then summarizes the code difference using two types of update primitives. Our experimental results showed that using update-conscious code update can greatly improve code similarity and thus reduce the update script sizes.

Categories and Subject Descriptors D.3.4 [*Processors*]: Code generation, Compilers, Incremental compilers, Interpreters, Optimization; K.6,3 [*Software Management*]: Software maintenance

General Terms Algorithms, Design, Experimentation

Keywords Incremental coalescing simple offset assignment (ICSOA), Incremental coalescing general offset assignment (ICGOA), context-aware script, context-unaware script

1. Introduction

Mobile embedded systems such as PDAs and cell phones widely integrate 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 processing of digital signals by integrating specially designed and optimized architectural components. For example, a dedicated address generation unit (AGU) can perform parallel address computation in *register-indirect-automatic* addressing mode. With *register-indirect-automatic* addressing, the memory address is stored in an address register (AR) whose value can be automatically updated within a small range before or after the memory access. The access 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 [12]. By carefully allocating variables in the memory, DSP compilers can generate efficient code with compact size and improved performance.

The problem of assigning variables in memory was formulated by Bartley [2] and Liao *et al.* [13] as simple offset assignment (SOA) when there is only one AR, and general offset assignment (GOA) when there are multiple ARs. Many heuristic algorithms have been proposed in the literature to reduce the code size and improve the performance [14, 24, 1, 22, 3, 9, 15, 26, 16, 21]. To keep low execution overhead and better performance, many DSP applications are optimized and then released in binary format.

DSP applications running on mobile embedded systems often need to be updated after their initial releases. Bug fixes are the most common need due to the increasing complexity of modern embedded applications. Another need is to upgrade the current application with new features. For example, the map service on iPhone currently has no voice instructions [7], a very useful functionality that might be added in the future.

Although it is possible to patch or upgrade the code by directly connecting the mobile system to a server e.g. using a USB cable, there are situations where using wireless or even more expensive satellite communication is the only choice. For example, people may be traveling or working in a wild field, and thus do not have wired access to the server or the Internet. Updating code through wireless communication tends to incur both energy overhead and pecuniary burden. For example, it consumes 1000 times more energy to communicate one bit than to execute one instruction under certain settings [18]. As another example, many wireless data plans charge per KByte fee if the traffic is beyond the monthly quota. Since mobile embedded systems can be recharged when users return to the home, the energy efficiency goal focuses more on reducing the amount of transmitted data, even at the cost of executing slightly more instructions.

It is challenging to achieve cost-efficient DSP code update through wireless communication. Although transmitting a *diff*-based update script instead of the complete new binary is an effective approach to reduce the overall communication overhead [23, 10, 5], studies showed that using existing update oblivious compilers, a small change at the source code level may result in significant binary changes [18]. Several schemes have been proposed to achieve energy efficiency in updating sensor code after deployment [23, 18]. However, they are not directly applicable to updating DSP code. A big difference between wireless sensors and mobile systems is that it is almost impossible to recharge sensor battery while periodically recharging mobile systems is very common. Preserving energy is much more critical in wireless sensor networks, e.g. running low quality code may draw more energy in

LCTES'10, April 13–15, 2010, Stockholm, Sweden.

the long run. For the compilation scheme to minimize update script size in [18], Li *et al.* proposed an update-conscious compilation approach to reduce update script size on systems that perform *base-register-plus-offset* addressing. This approach tries to improve the register allocation similarity when generating the new version, so that the update script is small. Unfortunately, for the embedded systems that intensively access memory through *register-indirect* addressing and use post-/pre- incremental automatic AR update, both the register allocation changes and the data allocation changes can cause instruction updates. Thus, this scheme is not applicable to the DSP applications.

In this paper, we propose an update-conscious offset assignment approach for minimizing *diff*-based script sizes in updating DSP applications through wireless or satellite communication. In particular, we observed that generating a better offset assignment plays an important role in determining the code size and performance of DSP applications. In a case study of different versions of a real DSPstone program, we found that the offset assignment might be significantly different after small changes at the source code level. We developed an incremental variable coalescing heuristic to improve code similarity before and after the update, and then designed two types of scripts to summarize the code difference using context-unaware and context-aware primitives respectively. We implemented and evaluated the proposed incremental offset assignment scheme. In the experiments, we observed that incremental assignment with context-aware update primitives greatly reduces the update script size for medium sized changes.

The remainder of the paper is organized as follows. Section 2 discusses the background. The update-conscious offset assignment scheme and update script generation are elaborated in Section 3 and 4 respectively. We extend the scheme to GOA in Section 5. The experiments are discussed and analyzed in Section 6. Section 7 discusses the related work and Section 8 concludes the paper.

2. Background

2.1 Auto addressing on DSP processors

The *address generation unit* (AGU) on DSP processors assists the address computation in parallel. For the most frequently used auto addressing instructions i.e. 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 two; otherwise an extra instruction is needed to update the address register (as shown in Figure 1). Extra addressing instructions increase code size and slow down the execution. Since allocating variables to different locations in memory affects the address distance of adjacent accesses, different offset assignment heuristics have been proposed to minimize the number of extra addressing instructions e.g.[13, 2, 14, 24, 1, 22, 3, 9, 15, 26, 16, 21].

The problem of assigning variables in memory was formulated as simple offset assignment (SOA) when there is only one AR, and general offset assignment (GOA) when there are multiple ARs.

Offset distance	1st memory access	2nd memory access
0	no	no
1	post	no
1	no	pre
2	post	pre
> 2	addr. update instr.	no
> 2	no	addr. update instr.

Figure 1. Addressing modes between two adjacent memory instructions.

2.2 Offset assignment with variable coalescing

In this paper we propose our scheme based on a recently proposed effective offset assignment heuristic using variable coalescing [26, 21]. The *coalescing simple offset assignment* (CSOA) scheme [21] builds up two auxiliary graphs: (i) an *access graph* in which each vertex denotes a variable, and the edge weight denotes the frequency of adjacent accesses of the two corresponding variables; (ii) an *interference graph* in which each vertex indicates a variable, and an edge between two vertices indicates the live ranges of these two variables overlap and these cannot be allocated into the same memory location.

The offset assignment problem is modeled as finding the maximum weight path cover on the access graph [13]. Since many variables have short live ranges, they can be allocated to the same location in memory. CSOA iteratively chooses an edge in the *access graph* and adds it to the maximum weight path, or coalesces two vertices that do not interfere with each other. The decision is made based on the cost/benefit equations of each choice.

3. Update-Conscious Offset Assignment

3.1 A motivational example

Figure 2 illustrates the motivation to design an update-conscious SOA. It requires two and six words respectively with and without variable coalescing. The one with variable coalescing requires no extra addressing instructions as the reduction of memory usage increases the likeliness of two adjacent memory accesses being close enough to avoid explicit addressing instructions.

After a small update of the code, i.e. the third instruction is changed (Figure 2(d)), recompiling the code using CSOA generates a very different variable coalescing result (Figure 2(h)). The memory layout difference further translates to selecting different addressing instructions at each memory access (Figure 3). Out of seven updated instructions, four of them are due to the data allocation change, i.e. column 5 in Figure 3. As a comparison, keeping these variables in their original positions requires two instructions to be updated, i.e. column 7 in Figure 3.

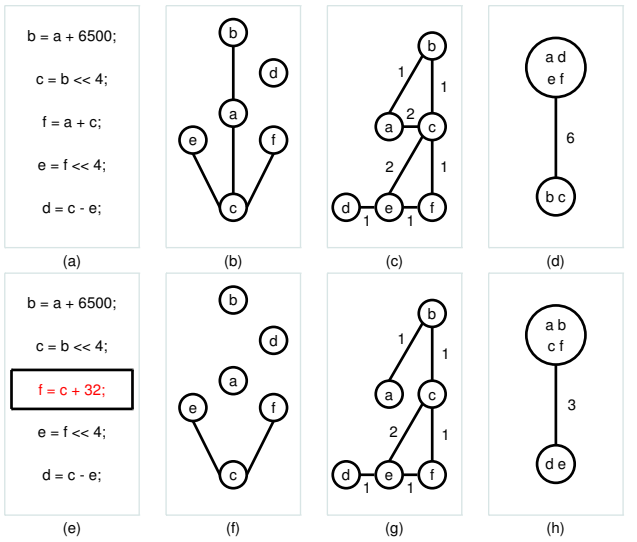

Figure 2. A motivational example: (a) the original C code segment; (b) the original interference graph; (c) the original access graph; (d) the offset assignment result using CSOA [21]; (e) the C code after a simple update; (f) the new interference graph; (g) the new access graph; (h) the new assignment using CSOA.

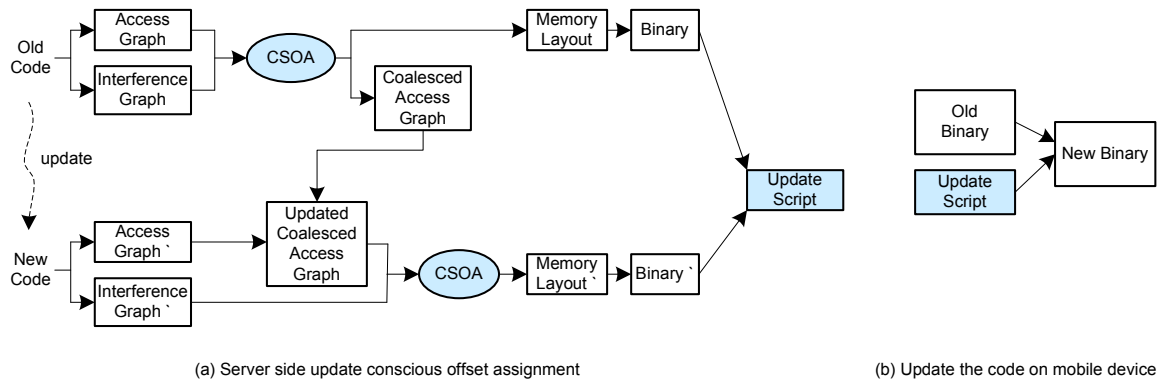

Figure 4. An overview of Incremental Coalescing Offset Assignment (ICSOA) -based code update scheme.

	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 3. Update script comparison between two versions using CSOA.

3.2 Incremental coalescing simple offset assignment (ICSOA)

To minimize the update script, we propose to perform update-conscious code updates through incremental coalescing SOA (ICSOA) (Figure 4). 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 1. It first builds the access graphs 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.

Function update_access_graph(). 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 big, 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.

Algorithm 1 Incremental Coalescing-Based SOA (ICSOA)

Input: AS_1,AS_2: access sequences before and after update;
IG_1,IG_2: interference graphs before and after update;
Output: the offset assignment.
1: $AG_1 \leftarrow$ Build access graph using AS_1;
2: $AG_2 \leftarrow$ Build access graph using AS_2;
3: $CAG_1 \leftarrow$ CSOA(AG_1,IG_1);
4: $AG_{NEW} \leftarrow$ update_access_graph(CAG_1,AG_2);
5: resolve_conflicts(AG_{NEW}, IG_2);
6: $CAG_2 \leftarrow$ CSOA(AG_{NEW}, IG_2);
7: Return offset assignment based on CAG_2;

Function resolve_conflicts(). Due to code update, two variables that are coalesced in the old assignment may interfere with each other. We identify this as a *conflict* and call *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 with other group members. The function then extracts the interfering variables with a higher factor one by one until all the interferences in the group are resolved. By doing so, the variables that create more interferences but have less adjacent accesses with others are extracted first 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 2, ICSOA combines the coalesced offset assignment (Figure 2(d)) and the new access graph (Figure 2(f)). Figure 5(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 5(c)). In this example, the script generated from ICSOA is 71% smaller than that of recompilation using CSOA.

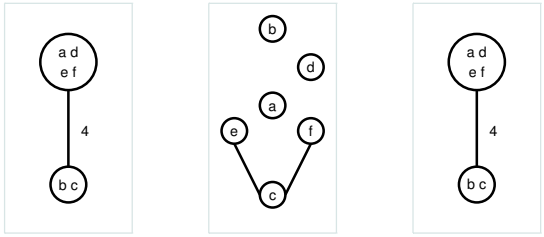

Figure 5. An example of ICSOA scheme: (a) AG_{NEW}, the updated access graph; (b) IG_2, the new interference graph; (c) the final offset assignment.

4. Code Update Script

After generating the new binary, our scheme summarizes its difference from the old binary in an update script that consists of a sequence of updating primitives specifying how to generate the new code on mobile devices (Figure 6). The primitives can be categorized to context-unaware or context-aware primitives, according to whether it will pass the updated memory layout to the remote devices.

4.1 Context-unaware script

The update script is generated by comparing the new and old binaries at the instruction level. To simplify the comparison, we link the unmodified code blocks in the old and new code, and then use the context-unaware primitives to specify how to update the changed code blocks.

4.1.1 Simple context-unaware primitives

There are four *simple primitives*: *insert*, *replace*, *copy* and *remove*. Both *insert* and *replace* primitives have a one-byte opcode and n-byte data or instructions to be inserted/replaced to the new code. The *copy* and *remove* primitives take one byte each and specify the size of data/instructions that need to be copied or removed.

To regenerate the new binary, the script interpreter on remote devices maintains two instruction pointers — one points to the old code and the other points to the last instruction that has been generated in the new code. The *insert* primitive inserts the instructions in its data part into the new code, and moves the pointer in the new code to the end. The *replace* primitive does the same thing to the new code but also moves the pointer in the old code for the same distance. The *copy* primitive reads the instructions from the old code, and moves both pointers.

	Primitive	Format and Operation	Size (bytes)
context-unaware	insert	000x xxxx (number of bytes to be inserted) data ... data	1 + number
	replace	001x xxxx (number of bytes to be replaced) data ... data	1 + number
	copy	010x xxxx (number of bytes to be copied)	1
	remove	011x xxxx (number of bytes to be removed)	1
	insert_access	100x xxxx (number of bytes to be inserted) data ... data	1+number
context-aware	copy_slot	110x xxxx (number of data slots to be copied)	1
	insert_slot	101x xxxx (number of data slots to be inserted) variable ... variable	1+number
	shift_slot	111x xxxx (number of consecutive unchanged slots) start_slot offset	3

Figure 6. Code update primitives.

4.1.2 Advanced context-unaware primitives

When inserting a new memory access between two existing accesses, we may need two *update* primitives and one *insert* primitive, as shown in Figure 7(d). Since the update primitives only modify the addressing modes, a compact way to express them 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 we introduce an *advanced context-unaware primitive – insert_access*.

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 7(c)), the new access is *c (located in memory slot 0)*, and the preceding memory access is *a (located in memory slot 1)*, so δ_{diff} is -1. Since it is the add operation that accesses *c* in the new instruction, the update primitive is

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

Rewriting the update script of the example, using the *insert_access* primitive, the script size is reduced by 50% (Figure 7(e)).

To maintain the correct program semantics, the interpreter has to know the memory locations accessed before and after the inserted instruction, and generate the correct addressing modes for these instructions. It is achieved by temporarily buffering each affected instruction, and updating its addressing mode when the memory location to be accessed in the next instruction is known.

4.2 Context-aware script

In our experiments, we observed that binary changes at several places may be caused by one memory layout change. For example, 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 summarizes an instruction level change. Instead, if the script interpreter on mobile devices can

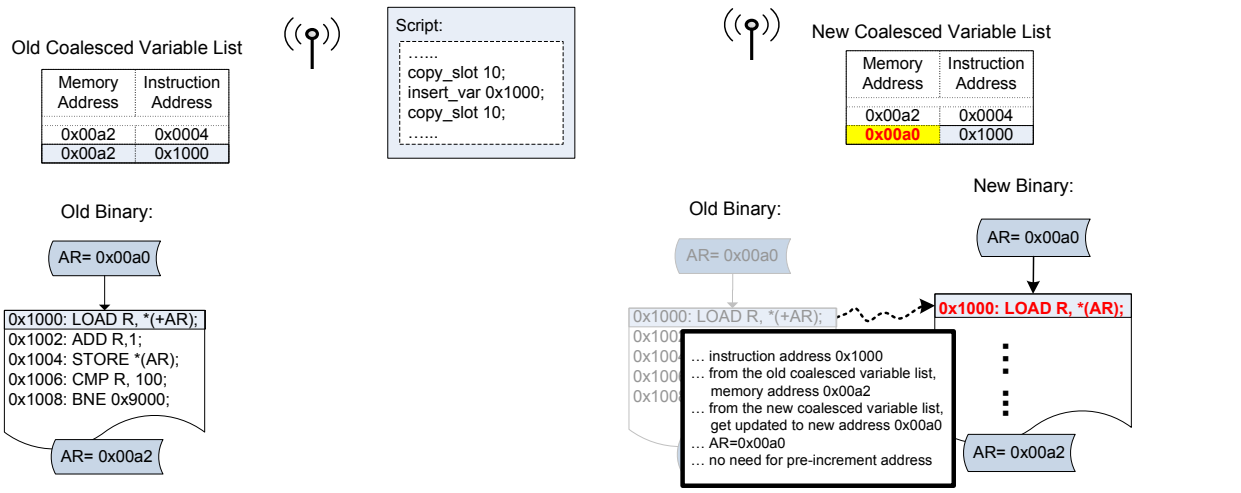

Figure 8. Context-aware code retrieval (The left shows the server side, and the right shows the mobile device side updates).

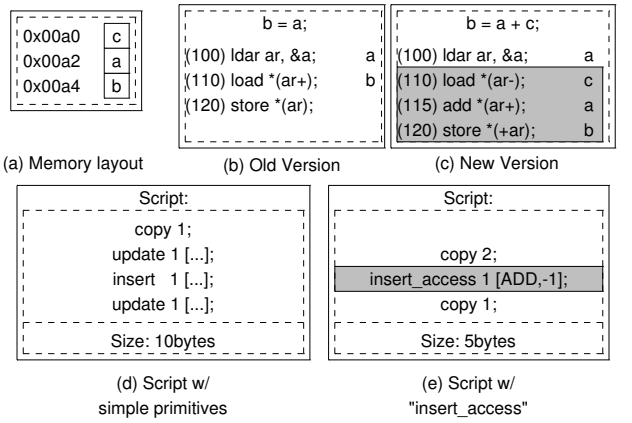

Figure 7. An example showing the use of *insert_access* primitive: (a) data allocation for both versions; (b) the original code; (c) the modified code; (d) update script using simple primitives; (e) update script using advanced primitives.

decode DSP instructions, and identify all its uses, then it is possible to send one "relocate `a`" primitive instead of individual instruction update.

We 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 *context-aware primitives* is to reduce the transmission of AMC instructions, and let the mobile devices construct them by themselves. Compared to the *insert_access* primitive, context-aware primitives are designed to update the code in more than one place.

4.2.1 Context-aware primitives

In order to update the AMC instructions automatically, the offset assignment changes (rather than affected instructions) need to be transmitted. Figure 6 lists the *context-aware* update primitives that we use to specify the memory layout change. We only consider the allocation of scalar variables in this paper. Each memory location contains one variable or multiple coalesced variables ([21, 26]).

copy_slot. This primitive copies multiple memory slots from the old offset assignment to the new assignment. There are two pointers pointing to the new and old assignments respectively. They are updated to the next slot with this primitive.

insert_var. This primitive adds a list of variables to the current memory slot in the old assignment. The related slot with the added variables is then copied to the new assignment. 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.

shift_slot. This primitive represents the case that multiple slots may be grouped and shifted from the old assignment to the new assignment. 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 Context-aware code retrieval

After receiving the update script, each sensor interprets the *context-aware primitives* to generate the new memory layout, and then interprets the *context-unaware 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 8, 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 8 illustrates our solution to this problem. We 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 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 context-aware 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.

4.2.3 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 [10] 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, in this paper we 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 we do not allocate any entry in the list. If multiple variables are coalesced and stored in the same memory location, we allocate the entries as follows.

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

Figure 9. Coalesced variable list.

Since the coalesced variables have their accesses spread in the code, we 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 9, 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 9 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 we 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. We 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 10.

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

Figure 10. The AR in/out value list.

The original list is preloaded on the mobile devices before deployment. The context-aware 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.

5. General Offset Assignment

In this section, we discuss our update-conscious offset assignment scheme in the presence of multiple address registers.

5.1 Coalescing general offset assignment (CGOA)

To compile a DSP application using k address registers, the CGOA scheme [21] first partitions all variables into k different sets. Variables in the same set use the same address register throughout the code. After partitioning the variables, the access graph and interference graph can be partitioned accordingly. The CSOA scheme is then applied to each access graph to generate the offset assignment for that address register. The overall offset assignment is the combination of individual assignments.

A brief discussion of variable partition is as follows. For each variable, CGOA computes a global interference number that is the number of interferences between this variable and all other variables. CGOA sorts the variables in decreasing order of their global interference numbers, and processes the variables iteratively. To decide which set a variable should be inserted, CGOA also computes the local interference number of this variable, i.e. the number of interferences with all the variables in each partition. CGOA assigns the variable to the partition with the lowest local interference number.

The objective of variable partition is to minimize the interference among variables assigned to the same set, and to increase the chances of variable coalescing in CSOA.

5.2 Incremental coalescing general offset assignment (ICGOA)

Our incremental variable coalescing based general offset assignment (ICGOA) scheme works as follows. It first divides the variables that exist in the old code into partitions based on their global interference numbers in the old code. It then sorts the new variables according to the decreasing order of their global interference numbers in the new code. ICGOA assigns new variables to the partitions according to their local interference numbers, similar to CGOA. After the variable partitioning, ICSOA is applied to each variable set to generate the offset assignment.

5.3 Update scripts

When generating the update script using only context-unaware primitives, there is no difference between ICSOA and ICGOA schemes.

When using context-aware primitives, we need to enhance the auxiliary data structures to handle each variable independently. Since variables in the coalesced variable list are sorted according to their memory addresses, and those using the same address register are grouped together, we only need to add a one-byte flag to terminate each group. That is, we need extra $k-1$ bytes for the architecture with k address registers. In addition we add information to the AR in/out value list to capture the entering/exiting values of

each address register. The update does not significantly increase the script size as the auxiliary data structures are preloaded, and only the modified sections are transmitted. With enhanced auxiliary data structures, ICGOA processes each address register independently similar to ICSOA as we discussed in preceding sections.

6. Experiments

6.1 Settings

We have implemented our proposed update-conscious ICSOA/ICGOA algorithms. We chose the Lance Compiler[11] to convert the source code (C code) into intermediate representations (IRs) from which the access sequence and interference graph are extracted. We selected the DSPstone[4] benchmark suite that is widely used to measure the performance of DSP compilers. We adopted CSOA-Offsetstone[20] as the baseline CSOA and implemented ICSOA on top of it.

We created the code update benchmarks using three methods:
i). compare two official releases;
ii). manually insert code changes to the application;
iii). automatically insert code changes to the application.
Then we generated the new binary using either CSOA/CGOA or ICSOA/ICGOA, and the update script using different scripting schemes. We evaluated their effectiveness based on code quality and code similarity.

6.2 Software update overhead

We manually made changes to two functions from DSPstone — the encoding/decoding verification function (*verify.c*) and the matrix multiplication function (*matrix1.c*). We modified these functions and, according to the impact on the existing code, categorized the changes into small and medium changes, according to the number of affected instructions (cases 1 to 5 in Figure 11).

In addition, one function may have different versions in DSPstone, such as the multiplication function *matrix.c* which has two versions, and the ADPCM standard implementation *opt_adpcm.c* which has four versions. We selected the *main* function in *matrix.c*, as well as the *speed_control* function in *opt_adpcm.c* as our benchmarks. The three manually generated test cases are divided into medium and large categories (cases 6 to 8 in Figure 11).

We evaluated the impact as the number of variable interferences that are added by the code update, and whether these 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.

Figure 12 compares the software update overhead for CSOA and ICSOA. We used three script formats to do the comparison.

- *Simple context-unaware script* that uses only the first types of context-unaware primitives;
- *Advanced context-unaware script* that uses all types of context-unaware primitives;
- *Context-aware script* that uses both context-unaware and context-aware 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 1, the code update is very small such that the difference between the old and new offset assignments is not big. We did not see much benefit using ICSOA over CSOA.

When comparing different script generators, we observed that between the two *context-unaware* schemes, the advanced *context-unaware* 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 4 and 5.

Test Case	Category	Function	Description
1	small	verify.c	Update **one basic block** to create the interference between two variables that are **not coalesced** in the original version.
2	medium	verify.c	Update **one basic block** to create the interference between three variables that are **coalesced** in the original version.
3	medium	verify.c	Expand the live ranges of three variables to **cross basic blocks**.
4	medium	matrix1.c	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.
5	medium	matrix1.c	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.
6	medium	matrix1.c ⇒matrix2.c	Move two iterations out of the loop.
7	medium	speed_control 1 ⇒2	Seven temporary variables are introduced to hold the value of the comparison results.
8	large	speed_control 2 ⇒3	Multiple global variables are combined into a structure. The reference to the variables are changed due to this change.

Figure 11. Experimental benchmarks.

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 4 and 5. We see a significant script size reduction by using this scheme. Adopting context-aware script tends to incur large complexity i.e. Test-Case 1 and 3 where we see a small script size increase due to the complexity to specify the offset assignment change. When the code has significant changes e.g. Test-Case 8 introduces 32% code changes, the old and new code segments are mixed such that the benefit from keeping the old data offset assignment diminishes.

GOA script size comparison. When there are multiple ARs, Figure 13 compares CGOA and ICGOA schemes with the different script generators.

When there are more ARs, recompiling the program results in large changes in both the variable partition and offset assignment. For Test-Case 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 big.

6.3 Code quality

In this paper we 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

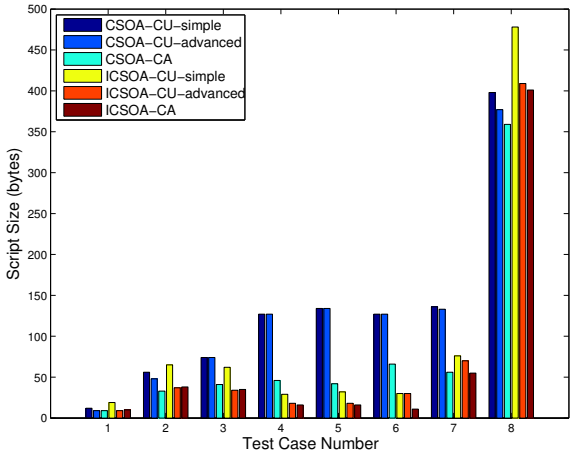

Figure 12. Script size comparison between CSOA and ICSOA (Single address register).

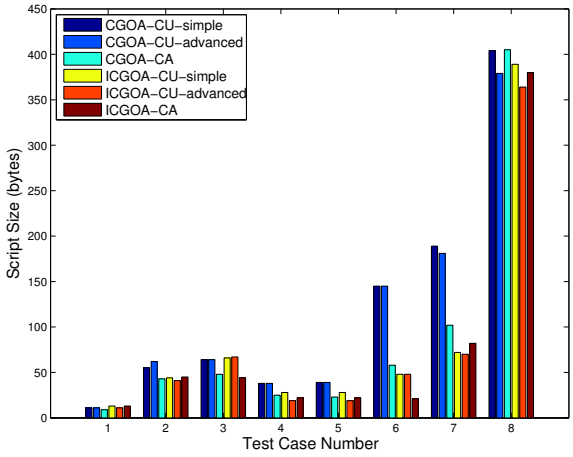

Figure 13. Script size comparison between CGOA and ICGOA (Double address register).

profiles. Although the latter provides more accurate evaluation, as we 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.

As shown in Figure 14, 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.

To better understand the code quality difference between two approaches, Figure 15 shows the breakdown of the execution overhead. We separated the new code at the intermediate representation (IR) level into the changed and unchanged parts. We then create their mapping to the binary level code segments.

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

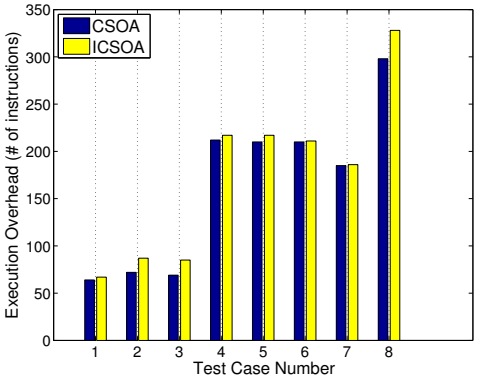

Figure 14. Code quality comparison between CSOA and ICSOA.

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

Figure 15. Execution overhead breakdown.

Figure 3; (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, we 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 15, we 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.

GOA performance comparison. For the test case 3 that has the largest code quality difference, we increased the number of available ARs, and found that with more available ARs, the code quality difference is reduced, as shown in Figure 16. The extra instruction number drops from 20% to 6% when the address register number is increased from 1 to 4. This is because with more ARs, the variables are partitioned into smaller sets. The software update tends to create less new interference and needs fewer patch variables. Fewer interferences result in less overhead in ICSOA.

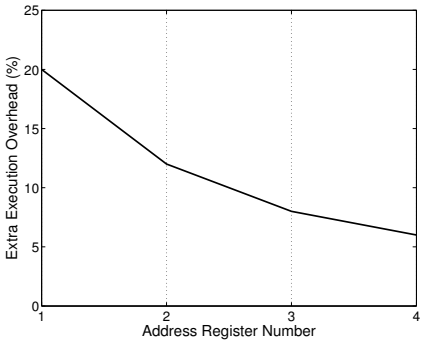

Figure 16. Code quality comparison with multiple ARs.

6.4 Random code insertion

We next inserted changes randomly into a file (*verify.c*) to study the robustness of our 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, we randomly generated 500 test cases and reported the average.

The script size comparison is shown in Figure 17. 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 we 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.

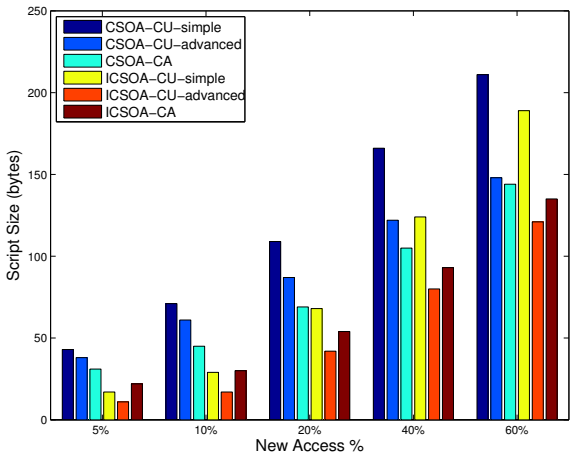

Figure 17. Script size comparison between recompile and incremental-compile (scattered random new code insertion).

The code quality is compared in Figure 18. Larger code update percentage, i.e. over 40%, has more live range extension of old variables, which produces more patch variables and instructions. Thus, the code produced by the recompilation scheme has a larger number of T4 type instructions; the code generated by the ICSOA scheme has a worse execution performance.

From Figure 18 and Figure 17, we 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 context-unaware advanced script is used, with about 2% extra instruction execution.

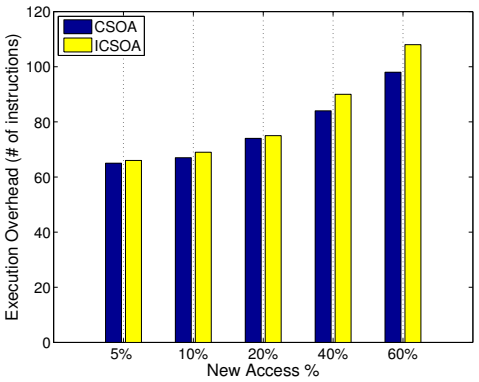

Figure 18. Code quality comparison between recompile and incremental-compile (scattered random new code insertion).

From the experimental results, we can also see that the context-unaware scheme works better with the incremental compilation scheme, and the context-aware scheme works better with the recompilation scheme. This is because that the context-aware scheme trades updating individual instructions for setting up the auxiliary data structures and letting the sensors to construct these updates. Thus, when we recompile the new version, a relatively large number of instructions are changed due to the data allocation differences, so the context-aware scheme can gain more benefit by saving those updates. On the other hand, when we use the incremental compilation technique, the saving is not great enough to balance the spending in setting up the data structures, therefore, the context-unaware scheme is more beneficial.

7. Related work

7.1 Software patching in resource constrained embedded systems

Software patching has become an integral part of software development, and is particularly important for systems that patch their code through wireless communication e.g. wireless sensor networks, and mobile embedded systems.

There have been efforts to design energy efficient post-deployment code dissemination in wireless sensor networks. Early schemes focused more on the protocol design and usually disseminated the entire new code [25, 6]. Recent schemes widely adopted the *diff*-based strategy. Reijers *et al.* [23] proposed simple update primitives to summarize the difference between new and old binaries, and disseminate the update script instead of the complete new code. Since without having the prior knowledge of code structure, Jeong *et al.* adapted the *rsync* algorithm to generate hashes of fixed code blocks from which the update script can be derived. Li *et al.* proposed update-conscious register allocation and data allocation techniques for applications using *base-register-plus-offset* addressing mode [18].

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. Marron *et al.* proposed to produce separate object files for TinyOS components linked by the sensor [19]. Dunkels *et al.* further proposed a dynamical linker for this system [5]. Koshy *et al.* proposed to relocated modules and generate the binary using a remote linker [10]. A drawback of releasing code not in the binary format is the increased runtime overhead, which might not be acceptable for tightly resource-constrained embedded systems.

We need algorithms that support efficient binary code update as DSP code is often highly optimized or even hand tuned to ensure performance and is released only in binary format. The update-conscious offset assignment algorithm in this paper is, to the best of our knowledge, the first such algorithm for DSP processors.

7.2 Offset assignment problem on DSP processors

Allocating variables to memory on DSP processors was formulated as offset assignment problem by Bartley *et al.* [2] and Liao *et al.* [13]. Liao *et al.* modeled the problem as finding the maximum weight path cover (MWPC) of the access graph [13]. Leupers *et al.* extended their work by proposing a *tie-breaking* heuristic in building access graphs and a variable partition heuristic for GOA [14]. Atri *et al.* improved the heuristic with an incremental SOA algorithm [1]. Sudarsanam *et al.* presented their algorithm [24] when the hardware supports auto addressing range [-L,+L]. Rao *et al.* proposed to reorder variables accessed in operations with commutative operands [22]. Choi *et al.* coupled offset assignment with code scheduling to minimize addressing instructions [3]. Kandemir *et al.* proposed more aggressive intra- and inter- statement transformation for reordering access sequences [9]. A genetic algorithm (GA) based algorithm was proposed by Leupers *et al.* [15] to handle multiple registers with addressing range [-L,+L]. Leupers constructed the Offsetstone benchmark suite [20] and conducted empirical comparison of major assignment heuristics [16]. A more comprehensive bibliography can also be found at this website [20].

Zhuang *et al* [26] and Ottoni *et al.* [21] independently developed algorithms to optimize offset assignment based on variable coalescing — variables that are not alive simultaneously can be allocated in the same memory location. They reported around 70% stack size reduction for both SOA and GOA. Our scheme is orthogonal to existing offset assignment heuristics and explores the offset algorithm space from a new direction.

8. Conclusions

In this paper we proposed an efficient code update scheme for achieving the best tradeoff between minimized update script size, compact binary size, and the runtime performance. Our study showed that, due to DSP code being closely coupled to data layout in memory, it is effective to perform incremental offset assignment such that the code similarity of the new and old code can be improved.

Acknowledgment

This work is supported in part by NSF CNS-0720595, NSF CAREER CCF-0641177. The authors thank anonymous reviewers for their constructive comments.The authors thank Dr. Jiang Zheng for the early discussion of the idea, and anonymous reviewers for their constructive comments.

References

[1] S. Atri, J. Ramanujam, and M. Kandemir, "Improving Offset Assignment for Embedded Processors," In *Languages and Compilers for High-Performance Computing*, S. Midkiff et al. (eds.), LNCS, Springer, 2001.

[2] D.H. Bartley, "Optimizing Stack Frame Accesses for Processors with Restricted Addressing Modes," In *Software: Practice and Experience*, 22(2):101–110, 1992.

[3] Y. Choi, and T. Kim, "Address Assignment Combined with Scheduling in DSP Code Generation," In *Design Automation Conference*, 2002.

[4] DSPStone Benchmark Suite, *http://www.iss.rwth-aachen.de/Projekte/Tools/DSPSTONE*.

[5] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt, "Run-Time Dynamic Linking for Reprogramming Wireless Sensor Networks," In *ACM International Conference on Embedded Networked Sensor Systems*, pp. 15–28, 2006.

[6] J. W. Hui, and D. Culler, "The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale," In *the 2nd ACM Conference on Embedded Networked Sensor Systems*, 2004.

[7] iPhone manual. http://www.apple.com.

[8] J. Jeong, and D. Culler, "Incremental Network Programming for Wireless Sensors," In *International Conference on Sensor and Ad Hoc Communications and Networks*, 2004.

[9] M. Kandemir, M. J. Irwin , G. Chen and J. Ramanujam, "Address Register Assignment for Reducing Code Size," In *the 12th International Conference on Compiler Construction*, 2003.

[10] J. Koshy, and R. Pandey, "Remote Incremental Linking for Energy-Efficient Reprogramming of Sensor Networks," In *European Workshop on Wireless Sensor Networks*, pp. 354–365, 2005.

[11] Lance Compiler *http://www.lancecompiler.com/*.

[12] P. Lapsley, J. Bier, A. Shoham, and EA Lee, "DSP Processor Fundamentals: Architectures and Features," Berkeley Design Technology, Inc., 1996.

[13] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang, "Storage Assignment to Decrease Code Size," In *ACM Transactions on Programming Language and Systems*, 18(3):235-253, 1996.

[14] R. Leupers, and P. Marwedel, "Algorithms for Address Assignment in DSP Code Generation," In *International Conference on Computer Aided Design*, pp. 109-112, 1996.

[15] R. Leupers and F. David, "A Uniform Optimization Technique for Offset Assignment Problems," In *International Symposium on System Synthesis*, pp. 3–8, 1998.

[16] R. Leupers, "Offset Assignment Showdown: Evaluation of DSP Address Code Optimization Algorithms," In *the 12th International Conference on Compiler Construction*, 2003.

[17] P. Levis, and D. Culler, "Mate: A Tiny Virtual Machine for Sensor Networks," In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 85–95, 2002.

[18] W. Li, Y. Zhang, J. Yang, and J. Zheng, "UCC: Update-conscious Compilation for Energy Efficiency in Wireless Sensor Networks," In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[19] P. J. Marron, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, and K. Rothermel, "FlexCup: A Flexible and Efficient Code Update Mechanism for Sensor Networks," In *European Workshop on Wireless Sensor Networks*, pp. 212–227, 2006.

[20] OffsetStone Benchmark Suite, *http://www.address-code-optimization.org*.

[21] D. Ottoni, G. Otoni, G. Unicamp, and R. Leupers, "Offset Assignment Using Simultaneous Variable Coalescing," In *ACM Transactions on Embedded Computing Systems*, 5(4):864-883, 2006.

[22] A. Rao, and S. Pande, "Storage Assignment Optimizations to Generate Compact and Efficient Code on Embedded DSPs," In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 128-138, 1999.

[23] N. Reijers, and K. Langendoen, "Efficient Code Distribution in Wireless Sensor Networks," In *International Workshop on Wireless Sensor Network Architecture*, pp. 60–67, 2003.

[24] A. Sudarsanam, S. Liao, and S. Devadas, "Analysis and Evaluation of Address Arithmetic Capabilities in Custom DSP Architectures," In *Design Automation Conference*, pp. 287–292, 1997.

[25] Crossbow Technology Inc. "Mote In-Network Programming User Reference," 2003.

[26] X. Zhuang, C. Lau, and S. Pande, "Storage Assignment Optimizations through Variable Coalescence for Embedded Processors," In *the International Conference on Languages, Compilers, and Tools for Embedded Systems*, 2003.