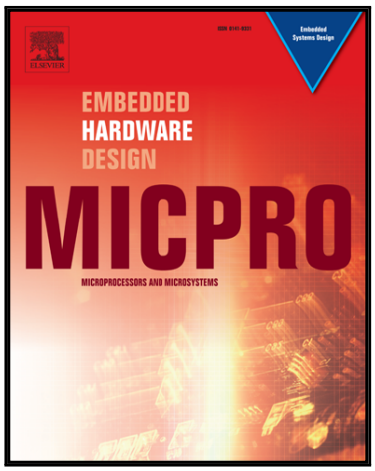


Region-based Dual Bank Register Allocation for Reduced Instruction Encoding Architectures

Je-Hyung Lee , Soo-Mook Moon , Jinpyo Park

PII: S0141-9331(17)30158-8
DOI: 10.1016/j.micpro.2017.09.005
Reference: MICPRO 2616

To appear in: *Microprocessors and Microsystems*

Received date: 10 March 2017
Accepted date: 20 September 2017

Please cite this article as: Je-Hyung Lee , Soo-Mook Moon , Jinpyo Park , Region-based Dual Bank Register Allocation for Reduced Instruction Encoding Architectures, *Microprocessors and Microsystems* (2017), doi: 10.1016/j.micpro.2017.09.005

Region-based Dual Bank Register Allocation for Reduced Instruction Encoding Architectures*

Je-Hyung Lee
Samsung Electronics
lordljh@gmail.com

Soo-Mook Moon
Seoul National University
smoon@snu.ac.kr

Jinpyo Park
Samsung Electronics
jinpyo11.park@gmail.com

Abstract

In embedded systems, small code size is important due to memory constraints. One technique to achieve a small code size is reducing the instruction encoding from 32-bit to 16-bit, such as the ARM THUMB or MIPS-16 architectures. This half-size encoding leads to shorter register operands, making fewer registers available for register allocation and causing more spills, although invisible registers can be used as spill locations via copies. We propose reconstructing the original register file into dual-banks, added with the bank toggle instruction for bank changes and the inter-bank copies between the banks. We also propose an efficient dual-bank register allocation technique based on regions in the code to reduce spills. As a case study, we applied our banked register allocation model for the THUMB architecture. We found that the code size decreases by as much as 8% (5.8% on average) while the performance improves by as much as 11.1% (3.3% on average). Our results indicate that we would better organize the register file of an embedded CPU that can provide reduced encoding into dual banks for better quality of register allocation, rather than using the invisible registers for spills.

Keywords: Reduced Encoding Architectures, Register Allocation, Banked Registers

1 Introduction

Compared to a general-purpose computer system, one of the most serious constraints of an embedded system is its limited memory. Since the memory price often dominates the whole price of an embedded system and it is almost impossible to expand the memory once the system is built, embedded software is always constrained by its code size. In fact, using small instruction memory may improve the power consumption of an embedded system, lengthening the battery life. This renders the optimizing compilers for embedded systems to focus on reducing code size than improving performance when there is a conflict between the two criteria, although small code size often leads to high performance.

In addition to the compiler optimization techniques, hardware techniques for reducing the code size have been introduced, the most popular one is reducing the instruction encoding. For example, the ARM THUMB [7] and the MIPS-16 [12] have a 16-bit instruction set instead of a 32-bit instruction set. This is achieved by reducing the bit width of the opcode as well as the bit width of register operands, as depicted in Figure 1, for the case of THUMB. With shorter instructions the same computation would require more instructions, so the instruction count increases, yet it is known that the code size decreases significantly due to its half-sized instructions, although the performance also decreases tangibly [7].

The shortened register operand fields for reduced encoding imply that fewer registers are available for register allocation, which can lead to higher register pressure and more spills, affecting both the code size and the performance negatively. For example, the ARM THUMB instructions have three-bit register operand fields instead of the four-bit fields of the original ARM instructions. So, only eight registers are available, while the processor still has sixteen registers. According to our observation, this limitation of registers leads to higher register spills than in the original architecture (see Section 5.2). Therefore, it is questioned if there is a way of using the unavailable registers, and one idea is employing an architectural mechanism called *banked register*.

* This is a revised and extended version of a paper published in the Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), Aug 2007 [17]. The major difference from the conference paper is that we focus on performance improvement as well as code size reduction of [17], by newly introducing inter-bank copies and by improving the quality of register allocation with better allocation. We also expanded the evaluation. This work was performed while Je-Hyung Lee and Jinpyo Park were at Seoul National University.

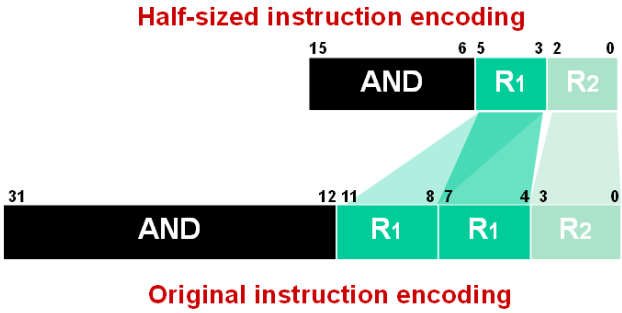


Figure 1. Reduced instruction encoding in the ARM THUMB

Generally, banked registers are a register file grouped into several banks, which have been used for various purposes in diverse contexts [5][6][8][9][14][16][17][18][19][20][21][23][24][26][38] (see Section 6). Our context of employing banked registers for reduced encoding architectures is reconstructing the original register file into *dual* banks and allowing only one bank to be active at a time with a *bank change* instruction. This can make all of the original registers available for register allocation including those otherwise unavailable, thus reducing the spill. This idea is also applicable to the originally compact encoding (8 or 16-bit) CPUs such as Motorola 68HC12 [26], when we want to double the number of registers without compromising its instruction encoding, by reorganizing the extended register file into dual banks [20].

To allocate banked registers, we need to partition the code into two regions, one for each register bank, and allocate registers separately from each bank. If there are variables live across regions, *inter-bank copies* should be inserted appropriately. The most important issue is how to partition the code efficiently so as to reduce the register pressure, hence the spills, while minimizing the bank changes and inter-bank copies. We propose an efficient heuristic for code partitioning and an elaborate region-based banked register allocation technique. Unlike previous techniques, our goal is reducing the code size as well as increasing the performance, so we try to reduce the bank change overhead while partitioning the code aggressively beyond basic blocks. We could obtain a competitive result for both the code size and the performance when we perform a case study with the THUMB, yet it is generally applicable to other reduced encoding architectures.

The contribution of this paper is as follows. We propose a banked register file for half-sized encoding architectures to utilize otherwise invisible registers and an efficient banked register allocation technique that reduces spills and bank change overhead. Our results provide a useful insight for an embedded CPU design such that if one wants to build one that can also provide the half-sized encoding feature, it would be desirable to organize the register file into dual banks for banked register allocation, rather than using the inaccessible registers as spill locations as in the THUMB.

The rest of the paper is organized as follows. In Section 2, we will show how to adopt banked registers in a reduced instruction encoding architecture. Section 3 briefly describes the ARM THUMB architecture and shows our architectural change with banked registers. In Section 4, we will explain the details of our banked register allocation technique. Section 5 reports our experimental results and Section 6 describes the related work. We summarize the paper in Section 7.

2 Banked Register Allocation for Reduced Encoding Architectures

In this section, we illustrate the benefit of banked register allocation with a simple example. We also provide a proposed banked register model for reduced encoding architecture and some intuition for banked register allocation.

2.1 An Example of Banked Register Allocation

To simplify the problem, we assume that the original architecture has four registers (`R0~R3`), while its reduced encoding version can access only half of them (`R0` and `R1`). The first column in Figure 2 shows a code example where four

variables (*a*, *b*, *c*, *d*) are live simultaneously, so four registers are required to hold all variables. The next column shows register-allocated code for the reduced encoding architecture without banked registers. Since only `R0` and `R1` are available, we cannot keep the four variables in registers, generating four spill instructions for *a* and *c* (two saves and two restores).

Banked register architectures require two special instructions. Since only one bank is available at a time (which is called an *active bank*), a *bank change* instruction is required to change the active bank. The architecture also needs an *inter-bank copy* instruction to copy registers between different banks. A bank change instruction can have extra bits available for performing a register copy as well, so we try to merge a bank change instruction with a copy (including an inter-bank copy), if possible. This can reduce the overhead of bank change instructions.

The last column in Figure 2 shows the register-allocated code with dual banks where two additional registers (`R2` and `R3`) are available for register allocation. The first bank change instruction copies `R1` to `R2` to use *b* defined in the initial bank at the new bank. The next bank change instruction changes the bank only (`mov R0←R0` does nothing). Although we generate two bank change instructions, we have no spill, saving two instructions compared to the original code.

Code before register allocation	Code after register allocation without banked registers	Code after register allocation with banked registers
define a	define R0 ← a	define R0 ← a
define b	**save R0** ; a	define R1 ← b
define c	define R0 ← b	**bank change & mov R2 ← R1** ; b
use b	define R1 ← c	define R3 ← c
define d	**save R1** ; c	use R2 ; b
use d	use R0 ; b	define R2 ← d
use c	define R1 ← d	use R2 ; d
use a	use R1 ; d	use R3 ; c
use b	**restore R1** ; c	**bank change & mov R0 ← R0**
	use R1 ; c	use R0 ; a
	restore R1 ; a	use R1 ; b
	use R1 ; a	
	use R0 ; b	

Figure 2. Example of reducing spills and code size by adopting banked register

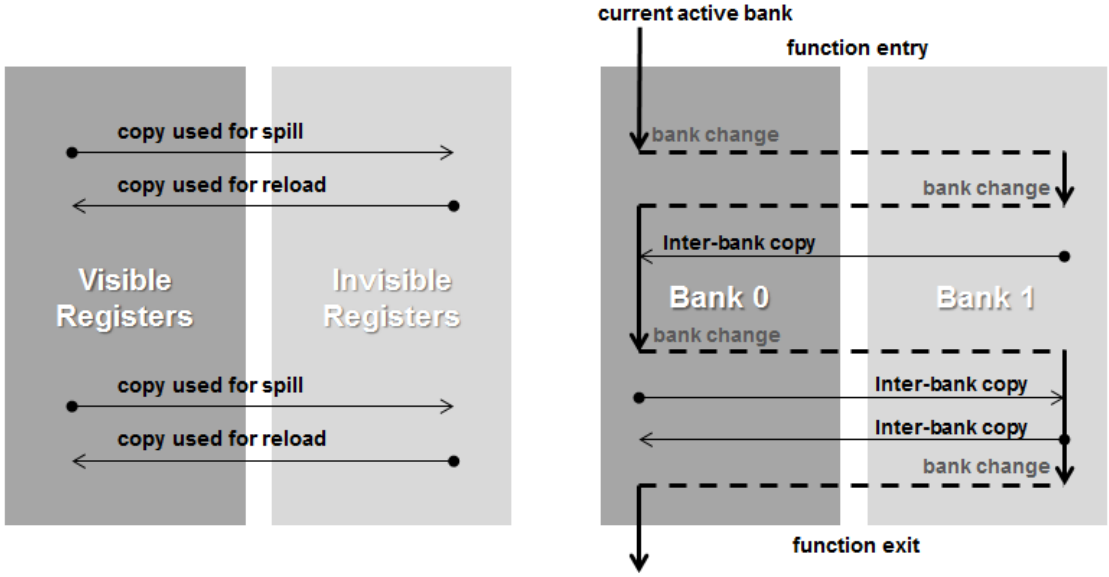
Figure 2 shows that the number of registers available for register allocation increases by employing register banks, yet there is an overhead of bank change instructions and inter-bank copies (although we could merge one copy with a bank change). The location for bank changes would be important for reducing spills and inter-bank copies, as we will see later.

2.2 Proposed Banked Register Model

Our proposed banked register model is depicted in Figure 3, compared to the original one. Figure 3 (a) shows the original register model where only half of the registers are available for register allocation. The other half is invisible, but only a few instructions including a copy can access them, as in the ARM THUMB. The THUMB exploits those invisible registers as spill locations such that copies to and from those registers are generated preferably over memory stores and loads, when spills are needed. Although the number of spill instructions is the same, thus no difference in the code size, the performance penalty associated with spills can be reduced. This means that the invisible registers are already being utilized in the original register model, not for reducing spills but for reducing their runtime overhead. It would be interesting to evaluate this approach compared to our approach proposed in this paper, which is using the invisible registers for reducing the spill itself by reorganizing the register file into dual banks.

Figure 3 (b) shows our proposed banked register model. It is a dual bank, composed of *bank0* and *bank1*, which correspond to the visible and invisible registers, respectively. Our bank change instruction actually *toggles* the active bank from one bank to the other instead of selecting a bank explicitly. To move data between the two banks, an inter-bank copy instruction is provided, which is similar to the copy to and from the invisible registers in the original register model. Such a

copy obviates the data movement using pushes and pops via memory (two instructions), which would cause a substantial overhead for both the code size and the performance. For a given function, the active bank at its entry should be identical to that at its exit. That is, if the active bank at the function entry is bank1, the active bank at the function exit should be bank1, and vice versa. This simplifies register allocation for function calls such that a function call can be made no matter what the current active bank is. We also require the calling convention (i.e., registers used for passing arguments and return values, caller-save/callee-save registers) to be identical for both banks.



(a) Original register model (b) Banked register model

Figure 3. Banked register model

Compared to the original register model, the proposed banked register model may reduce spills, which would be advantageous in reducing the code size, but not necessarily due to the overhead of bank changes and inter-bank copies. Also, the proposed model is not always advantageous from the performance perspectives since spills are now composed of loads and stores only, unlike in the original model where spills may include register copies to and from the invisible registers. So, restructuring into banked registers does not exactly mean doubling up the number of available registers, and we need an elaborate algorithm for efficient banked register allocation.

2.3 Some Intuition for Banked Register Allocation

Banked register allocation requires partitioning the code into two regions that use different register banks. Each region can access only one register bank and there are bank toggles at the region boundaries. Region partitioning involves partitioning all live ranges in the code into two groups. Those live ranges defined and used within a region should be allocated to the corresponding register bank. For those live ranges that are live across two regions (e.g., defined in one region but used in the other region) we split them at the region boundary using inter-bank copies such that one split belongs to one bank while the other split belongs to the other bank. After all (split) live ranges are partitioned into two groups in this way, we employ an existing global register allocation technique separately for each group as if it is a bank-less environment. The most important issue in this process would be how to partition regions efficiently.

For a given function, we call the bank at the function entry the *primary bank* and the other bank the *secondary bank*, respectively. A region allocated to the primary bank is called a *primary region* while a region allocated to the secondary bank is called a *secondary region*. For example, if a function is invoked at bank1 at runtime, its primary bank is bank1 and its secondary bank is bank0, and the function's primary regions and secondary regions use bank1 and bank0, respectively.

Before we partition regions, the whole function is regarded as belonging to a single primary region. Now, we choose the code regions that belong to the secondary regions in a way of reducing the register pressure of this primary region. A good candidate is a code region where the register pressure is high such that allocating only to the primary bank would lead to the spills. Such a region should include at least one variable whose live range is defined and used completely within the region (we call it a *region-local* variable). We need at least one region-local variable for a secondary region, since it can be allocated to the secondary bank completely, which would reduce the register pressure of the primary region, hence the spills. After we identify the secondary regions, the remaining regions will become the primary regions.

In addition to reducing the register pressure, we also need to consider the bank change overhead when choosing the secondary regions. That is, *bank change instructions* need to be added at the region boundaries, which will be an overhead. Also, for those variables partially contained in the region (i.e., some of their uses or definitions are outside the region, which we call *region-across* variables), *inter-bank copies* need to be added to transfer their values between the regions.

Region-local variables and region-across variables for a secondary region are illustrated in Figure 4. There can also be *region-thru* variables which pass through the region, yet being defined and used entirely outside the region. Region-thru variables will be allocated entirely to the primary bank, hence not being considered when allocating the secondary region.

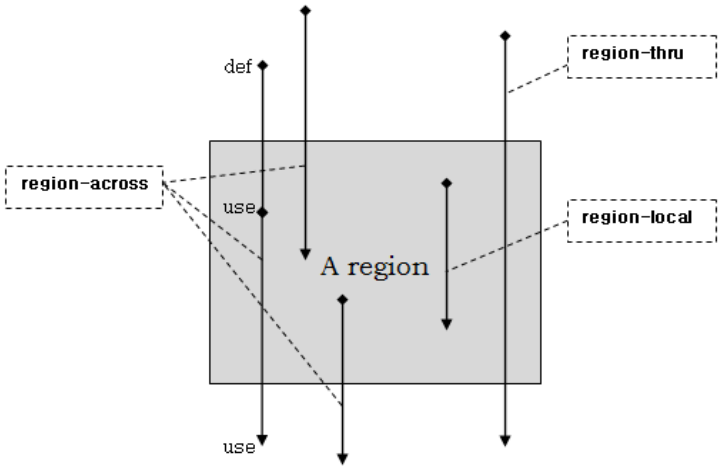


Figure 4. Classifying variables in a secondary region

3 Banked Registers for the ARM THUMB Architecture

In this section, we describe how to apply a banked register model for a given reduced encoding architecture, using the ARM THUMB as an example. We first summarize the original THUMB architecture, followed by our proposal for restructuring its register file into a banked register file. The focus of this paper is not about architectural modification of the THUMB, but about banked register allocation model, and we use the THUMB as a target of the case study. As such, we do not deal with its micro-architectural complexity since it is beyond the scope of this paper.

3.1 Summary of the ARM THUMB Architecture

The THUMB is a special instruction set extension introduced first in the ARM7TDMI processor [3]. It takes 16-bit instruction encoding, yet works in the 32-bit ARM processor [7]. The THUMB instruction set is a complete subset of the ARM instruction set, so a THUMB instruction is directly translated to a corresponding ARM instruction by an embedded de-compressor before entering the execution stage of the ARM pipeline.

The THUMB is based on two-operand instructions (e.g., AND R1, R2) instead of three-operand ones (e.g., AND R1,

R1, R2) in the ARM[1], which means that the THUMB needs more instructions for the same computation. Also, the THUMB register operand fields have three-bits compared to four-bits in the ARM, which may lead to more spills due to fewer registers available. Nevertheless, the THUMB's code size is known to be around 30% smaller than the ARM's due to its half-size encoding, yet its performance is around 20% lower due to its higher instruction execution counts [7][16].

Among the original 16 registers of the ARM, THUMB can use only 8 registers (`R0-R7`) for general purpose. `R13`, `R14`, and `R15` are special-purpose registers as in the ARM; `R13` for stack pointer (`SP`), `R14` for link register (`LR`), and `R15` for program counter (`PC`). The remaining 5 registers, `R8-R12`, cannot be used for general purpose, but can be used in special instructions (`add`, `mov`, `cmp` and `bx`) known as *format 5* instructions [3]. Many compilers employ these five registers as spill locations using `mov`[2], which can reduce the memory access overhead. However, from the perspective of code size, both spill to the memory and spill to registers are equivalent. So, we want to reduce the spill itself by allocating those otherwise unavailable registers using banked registers, which will reduce the code size and improve the performance.

3.2 Architecture Modification for Banked Registers

The proposed *banked-THUMB* (b-THUMB) has a dual bank composed of the bank0 (`R0-R7`) and the bank1 (`R8-R15`). To specify the active bank, we assign one-bit bank specifier in the ARM's *current program status register*, which is combined with the register operand number to determine the real register number at runtime. We add a bank change instruction, `mvb`, which toggles the bank specifier to change the active bank. This instruction has two register operands like the THUMB copy instruction `mov`, so `mvb` can copy registers as well as toggle the active bank. Inter-bank copies are added for region-across variables. Since the THUMB `mov` instruction can already access all ARM registers without any restriction, no modification is required to use it for inter-bank copying in our b-THUMB architecture.

The banked registers should be completely *symmetric* in the sense that the bank0 and the bank1 must be identically configured. In our b-THUMB architecture, the bank1 includes three special registers `R13`, `R14`, and `R15`, as in the THUMB, which break the symmetry. To make symmetric dual register banks, we separate the three special registers, `SP`, `LR`, and `PC`, from the register file and assign `R13`, `R14`, and `R15` as general-purpose registers. This asymmetry would not occur in the MIPS-16 based banked registers since its special-purpose registers are separated from R0-R15.

With our bank toggling approach, banked register allocation is simplified due to the bank symmetry since we do not have to know what the current active bank is when allocating registers. For example, if the symmetry were not satisfied as in the original THUMB architecture, `mov R7,#1` would assign a constant 1 to the register `R7` when in bank0, but would assign 1 to the `PC` when in bank1, causing an error. Also, we can have the same calling convention with the symmetric banks, irrespective of the current bank. In our b-THUMB calling convention, the arguments and the return value are allocated to (`R0-R3`) and (`R0,R1`), respectively, when in bank0, while they are allocated to (`R8-R11`) and (`R8,R9`), respectively, when in bank1. This follows the original THUMB calling convention (except for caller/callee saves) [3].

Unfortunately, our symmetric banks might make the comparative evaluation of the original THUMB and the proposed banked THUMB somewhat unfair, since the banked THUMB appears to have more general-purpose registers. However, the evaluation is completely fair from the code size perspectives, and from the performance perspectives, there is negligible difference, as will be explained in our experimental results (see Section 5.4).

One advantage of bank toggling is that we can start a function at any bank, as long as we finish the function at the same bank as at the entry. This is more flexible than requiring a function to start execution always at bank0, for example, which will enforce a function call to be made only when the current bank is bank0; additional instructions including bank changes might be needed to make the call be located inside bank0 and to move arguments and return value across the bank changes.

It might not be straightforward to understand the register numbers in the b-THUMB code since its three-bit register field

[1] In the THUMB ISA and the ARM ISA, the destination register comes first, followed by source registers.

[2] Other format 5 instructions rarely use `R8-R12`, and it is strictly disallowed in our THUMB code generation so that those invisible are

can have only R0-R7, while it can really mean accessing R8-R15 as well as R0-R7, depending on the current active bank. Table 1 illustrates an example of how register numbers are expressed when the b-THUMB assembly code is generated, assembled, and run in our tool chain. In the example C code in Table 1 (a), there are two function calls to **foo()**, and we assume that there is a bank change between the two calls. Here, the function **foo()** has two arguments and a return value, which are supposed to be allocated to (R0, R1) and (R0), respectively, when in bank0, while they should be allocated to (R8, R9) and (R8), respectively, when in bank1.

Table 1 (b) shows the b-THUMB assembly code generated by our register allocator, where we simply allocate the primary region with R0-R7, while allocating the secondary region with R8-R15 (so R0-R7 and R8-R15 are regarded as the primary bank and the secondary bank, respectively). So, we allocate the arguments and the return value to (R0, R1) and (R0) for the first call, and allocate them to (R8, R9) and (R8) for the second call. There is an mvb merged with an inter-bank copy between the banks. It should be noted that this assembly convention is just for better readability, and it will be decided at runtime which bank (bank0 or bank1) will actually be the primary or secondary bank.

Table 1. Register Numbers

(a) C source	(b) Assembly code generated by our banked register allocator	(c) Assembled object file	(d) Run time (entry is in bank0)	(e) Run time (entry is in bank1)

b=foo(1,a);
 	
c=foo(12,b)+2;
	mov R0, #1
	ldr R1, [SP]
	call foo
	mvb R9, R0 ;bank toggle
	mov R8, #12
	call foo
	add R8, #2
		mov R0, #1
		ldr R1, [SP]
		call foo
		mvb R9, R0
		mov R0, #12
		call foo
		add R0, #2
			mov R0, #1
			ldr R1, [SP]
			call foo
			mvb R9, R0
			mov R8, #12
			call foo
			add R8, #2
				mov R8, #1
				ldr R9, [SP]
				call foo
				mvb R1, R8
				mov R0, #12
				call foo
				add R0, #2

Our assembler converts this code into a legitimate one in Table 1 (c), which includes only R0-R7, except for mvb which uses R9. Grey-colored R0-R7 in Table 1 (c) will address the bank opposite from the bank where we start. Table 1 (d) and (e) show the real register numbers during runtime when we start the code in bank0 and bank1 (i.e., the primary bank is bank0 and bank1), respectively. If we start in bank1, for example, R0-R1 in the first call in Table 1 (c) actually addresses the real R8-R9 while R0-R1 in the second call actually addresses the real R0-R1, as shown in Table 1 (e). So wherever the function **foo()** is called (either at bank0 or at bank1), the allocated code will work, and this is possible due to the bank symmetry and the bank toggling mechanism.

4 Region-based Banked Register Allocation

Previous sections described our banked register extension for the THUMB and some intuition for region-based banked register allocation. In this section, we describe our register allocation technique in detail with an example. Basically, banked register allocation requires partitioning the code into two regions and allocate each using a different register bank, with inter-bank copies added for those live ranges across regions. Good partitioning is needed for good register allocation.

For our example, let us assume that the original architecture has 8 registers (R0-R7) instead of 16, and its reduced encoding THUMB has only 4 registers (R0-R3) available. R4 is used as a temporary spill location and R5-R7 are used for special purpose registers (SP, LR, PC). All registers are caller-save and arguments are passed via R0-R3. Our banked-THUMB (b-THUMB) architecture has dual register banks, with the bank0 composed of R0-R3 and with the bank1 composed of R4-R7. Those three special-purpose registers are assumed to be located separately.

Figure 5 (a) shows an example C function **foo()** and Figure 5 (b) is its original THUMB assembly code. There are

---

used as spill locations only.

four arguments (x, y, v, w) which are live throughout **foo()**, requiring four registers (R0, R1, R2, R3 in this order), yet the *else* block needs additional two registers to keep temporary calculation results, as shown in the .L2 block of Figure 5 (b). Therefore, the two registers R2 and R3, allocated to v and w, respectively, are spilled as shown in Figure 5 (b) (R2 is spilled to the register R4, while R3 is spilled to the memory). Consequently, four spill instructions are generated, with two additional stack adjustment instructions (sub SP,SP,#4 and add SP,SP,#4) to allocate a spill area in the stack.

Figure 5 (c) is the corresponding b-THUMB code with banked registers. We choose a secondary region (depicted with a shade) and add mvbs at its boundaries. The temporary calculation results in the .L2 block are now region-local variables of the secondary region and are allocated to R5 and R6. This allows v and w to be kept in their original registers (R2 and R3) without any spill. Since there is no spill, the stack adjustment instructions are also obviated.

For the region-across variable x, we add copies between the primary region (R0) and the secondary region (R4) at the region boundaries. Those inter-bank copies are successfully merged with the bank change instructions, as mvb R4, R0 and mvb R0, R4, as shown in Figure 5 (c).

```
int foo(int x, int y, int v, int w)
{
    if(x != y) {
        x = x + 1;
    } else {
        x = ((x & 0x100) << 1)
        + ((x & 0x2) << 1) ;
    }
    return (x + y + v + w);
}
```

(a)

```
    push    {LR}
    sub     SP, #4
    mov     R4, R2      ;spill
    str     R3, [SP]    ;spill
    cmp     R0, R1
    beq     .L2
    add     R0, R0, #1
    b       .L3
.L2:
    mov     R2, #1
    lsl     R2, R2, #8
    mov     R3, #2
    and     R2, R0
    and     R3, R0
    lsl     R2, R2, #1
    lsl     R0, R3, #1
    add     R0, R2
.L3:
    mov     R3, R4      ;spill
    add     R0, R1
    add     R0, R3
    ldr     R3, [SP]    ;spill
    add     R0, R3
    add     SP, #4
    pop     {PC}
```

(b)

```
    push    {LR}
    cmp     R0, R1
    mvb     R4, R0    ;bank change
    beq     .L2
    add     R4, R4, #1
    b       .L3
.L2:
    mov     R5, #1
    lsl     R5, R5, #8
    mov     R6, #2
    and     R5, R4
    and     R6, R4
    lsl     R5, R5, #1
    lsl     R4, R6, #1
    add     R4, R5
.L3:
    mvb     R0, R4    ;bank change
    add     R0, R1
    add     R0, R2
    add     R0, R3
    pop     {PC}
```

(c)

Figure 5. An example of reducing spills with banked registers

Given the example, we will explain the detailed procedure of our region-based banked register allocation. In the current implementation, the input is the THUMB assembly code of a function as in Figure 5 (b) and the output is the b-THUMB assembly code as in Figure 5 (c). We do not generate the banked-register code directly by modifying the compiler backend; instead we perform a kind of binary translation by converting an existing THUMB assembly code to the b-THUMB assembly code for fast implementation. Our translator is composed of seven phases as follows:

(1) Detecting Spills: We first check if there is any spill code for a given function. If there is no spill, we do not perform anything. In the THUMB code, spills mean stores/loads to and from the stack (which we call *memory spills*) or register copies to and from the invisible registers (which we call *register spills*). Since some of the accesses to the stack are for parameter passing or array accesses, we identify the genuine memory spills by checking the stack offsets of loads/stores.

(2) Renaming: We perform control and data flow analysis to identify live ranges and transform them into pseudo registers. Memory spills are transformed to register copies by introducing new pseudo registers corresponding to their stack offsets. For example, the transformed pseudo code for def R1; store R1 @spill; ...; R2=load @spill; use R2 is as follows:

`def x`; copy `s=x`; ...; copy `x'=s`; `use x'`. These "spill pseudo registers" for memory spills (`s` in this code) or "invisible pseudo registers" for register spills are remembered so that if spills are required later in the register allocation phase, these pseudo registers are preferred for the actual spill. This is much simpler than generating the spills from scratch. We can also avoid any new spills that do not exist in the original THUMB code, so that we are likely to generate shorter code than the THUMB code.

Figure 6 depicts the control flow graph corresponding to Figure 5 (b) after introducing pseudo registers, which start with `S` combined with a number starting from 8. Reserved registers for argument passing (`R0-R3`) and return value (`R0`) are not changed. The numbers listed in `{}` are live variables flowing into each instruction (*live-in* variables), and they will be used for detecting basic blocks (BBs) that have a high register pressure, which will be the *seed* BBs for the secondary regions.

(3) Finding Seed Basic Blocks for the Secondary Regions: As we mentioned previously, we should place secondary regions where the register pressure is high. Moreover, a secondary region should have at least one region-local variable, which can be allocated to the secondary bank completely, reducing the register pressure of the primary bank.

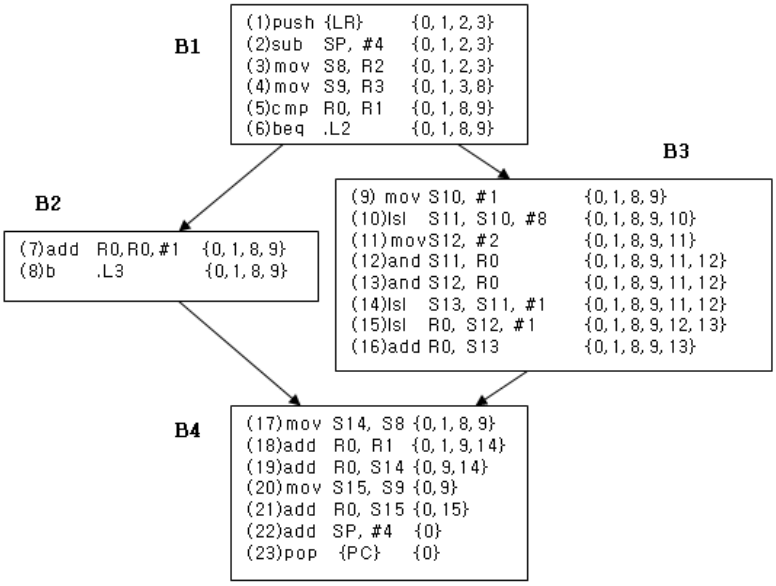


Figure 6. Control flow of the example code in pseudo form with live-in information

As a starting point of a secondary region, we define a *seed basic block* (SBB) which satisfies the followings: (a) it is a BB that includes an instruction whose number of live-in variables is bigger than K (the number of registers in a bank; K=4 in our example), and (b) there is at least one region-local variable within it. In Figure 6, `B3` can be an SBB since there are instructions which have six live-in variables and there are four region-local variables (`S10`, `S11`, `S12`, and `S13`).

We first identify all SBBs within a function. Then we expand those SBBs so as to define a larger region based on multiple SBBs. This will expose more region-local variables that can be allocated to the secondary bank, thus leading to a better chance of reducing the register pressure of the primary bank, and hopefully, reducing the spills as well. Moreover, we can reduce the number of bank changes or inter-bank copies by merging multiple single-SBB-based regions.

(4) Expansion to Enclosing Block: We expand an SBB into an *enclosing block*, which is defined as a single-entry single-exit (SESE) block where there is a single entry BB and a single exit BB [10]. One difference from the conventional SESE block is that if there is a backward edge flowing into the entry BB, it must come from the exit BB. Similarly, if there is a backward edge flowing out of the exit BB, it must go to the entry BB. For example, both Figure 7 (a) and (b) are SESE blocks, yet Figure 7 (b) includes a backward edge to the entry block coming from a non-exit BB, thus not an enclosing

block. Only Figure 7 (a) satisfies our enclosing block requirement.

Our enclosing block simplifies defining secondary regions such that placing a pair of bank toggle instructions (one in the entry BB and one in the exit BB) would simply make the region between them a secondary region. In Figure 7 (a), we place a pair of bank toggle instructions and the shaded region becomes a secondary region, whose bank is the secondary bank even while the loop iterates. On the other hand, if we place a pair of bank toggle instructions in Figure 7 (b), the shaded region cannot be a secondary region since its bank will be toggled every time when we iterate through the problematic backward edge. Our definition of enclosing block excludes such a faulty, secondary region.

The expansion process works recursively. Each SBB is an enclosing block by itself. For a given enclosing block E, we expand it to a larger enclosing block E' which is the *smallest* enclosing block that includes E. More specifically, we merge E with a predecessor BB of its entry BB (or a successor BB of its exit BB) and then try to find the smallest enclosing block E' that includes the merged E. This expansion process terminates when E' is estimated to be too "large" to be allocated to the secondary region. The estimation is performed as follows. We define a tentative secondary region in E' which is E' minus the entry and the exit BBs. We check if the number of concurrent, region-local live variables somewhere in this tentative region exceeds K; if so, we would definitely need spills in the secondary region, which we want to avoid (even if the number does not exceed K, it does not necessarily mean that we can avoid spills, but we would have a better chance).

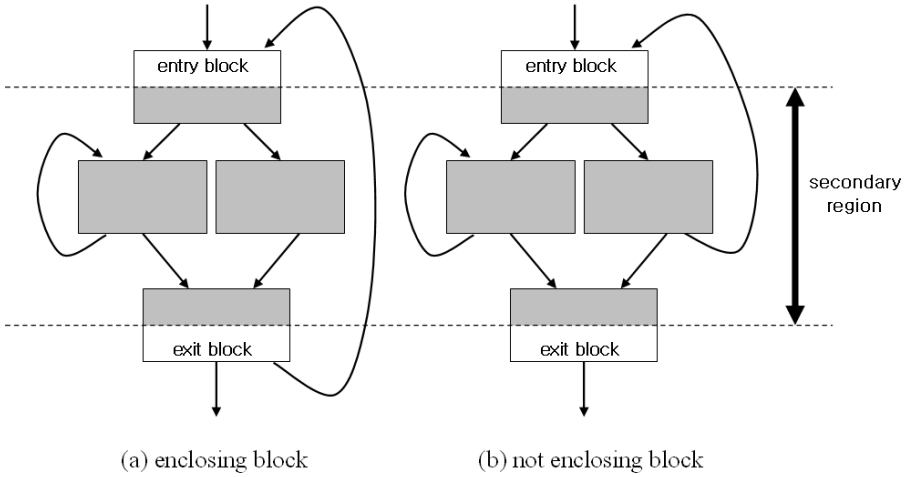


Figure 7. Example of enclosing block and non-enclosing block

The reason why we exclude the entry and the exit BBs is that we can adjust the final location of the bank toggle instructions within these BBs, so even if the number exceeds K somewhere in those BBs, we can find a location to place a bank toggle instruction within these BBs such that the number of concurrent region-local variables does not exceed K anywhere in the resulting secondary region (we can at least place a bank toggle instruction at the bottom of the entry BB and at the top of the exit BB). Choosing the exact bank toggle locations will be discussed in the next phase.

In our example in Figure 6, `B3` is the only SBB. It has a predecessor `B1` and the smallest enclosing block that includes both `B1` and `B3` is {`B1, B2, B3, B4`}. The maximum number of region-local, concurrent live variables in the tentative region composed of `B2` and `B3` (excluding the entry and the exit BBs) is two at the instructions in [(12), (15)] (where the variables 11, 12, and 13 are region-local variables while variables 0, 1, 8, and 9 are not). This does not exceed K (=4) in our example. Therefore, the enclosing block {`B1, B2, B3, B4`} is a legitimate one, meeting our expansion policy.

(5) Selecting the Locations for Bank Change: For each enclosing block computed in the previous stage, we finalize the secondary region by placing a bank toggle instruction in the entry BB and in the exit BB. We want to achieve the best region that can minimize the code size. For this, we estimate the cost and the benefit for each possible candidate region. The cost is the number of additional instructions needed including the bank toggle instructions and inter-bank copies. The

benefit is the number of spill instructions that can be removed due to the allocation of variables to the secondary bank.

The cost can be computed almost precisely: (1) two bank change instructions are required, (2) inter-bank copies are needed for all *region-across live-in* variables which are defined outside of the region but are used inside the region, and for all *region-across live-out* variables which are defined inside the region but are used outside the region. Since a bank toggle instruction is merged with one inter-bank copy, its cost is ignored if there is any inter-bank copy at the region boundary.

Estimating the benefit at this point is not easy since the removal of spills is decided in the next stage when we actually perform register allocation. We approximate it based on the number of concurrent region-local variables, expecting that allocating them to the secondary bank would allow otherwise spilled variables to be allocated to registers.

In our example in Figure 6, the smallest secondary region would be [(6), (17))[3] where the bank toggles are located at the bottom of the entry BB and at the top of the exit BB. The largest region would be [(3), (21)] by placing a bank toggle instruction just before (3) and placing another after (21) (we normally exclude the function prolog and epilog from the secondary region, so (1), (2), (22), and (23) are not included in the region for this example).

Table 2 shows the cost estimated for some candidate regions of Figure 6. It includes the number of region-across live-in and live-out variables, for each of which we need an inter-bank copy. We also add the cost of bank change instructions, which is zero for the candidate regions in Table 2 since all of them can be merged with inter-bank copies. We choose a region with the smallest cost, which corresponds to [(6), (17)) (there is no problem in placing the toggle instruction between a compare (5) and a branch (6), since the toggle instruction does not change the condition flag).

Table 2. Estimated cost for a few candidate regions

Region	Region-across live-in variables	Region-across live-out variables	Cost of inter-bank copies	Cost of bank change instructions	Total cost
[(6), (17))	R0	R0	**2**	**0**	**2**
[(6), (17)]	R0,S8	R0,S14	**4**	**0**	**4**
[(5), (17))	R0,R1	R0	**3**	**0**	**3**
[(5), (17)]	R0,R1,S8	R0,S14	**5**	**0**	**5**

(6) Making Interfaces between Regions: After we decide the secondary regions, we add bank toggle instructions at the region boundaries and add inter-bank copies. If there is a copy adjacent to a bank toggle instruction, we can merge them.

There is an issue on where to place the inter-bank copies. An inter-bank copy actually serves as *splitting* a region-across variable into two live ranges, one for the primary region and the other for the secondary region (which we call the *primary split* and the *secondary split*, respectively), such that each split can be allocated to a different bank. There are three approaches to placing inter-bank copies, which we will explain in Figure 8 for two region-across variables, s100 and s101.

One is simply placing all inter-bank copies at the region boundaries. One problem of this approach is that the secondary split live range might be too long, increasing the register pressure of the secondary region. For example, if we put inter-bank copies at the region entry in Figure 8 (a), their secondary split live range span from the definition of the copy at the region entry to its uses, thus likely to interfere more with other region-local variables.

Another approach is placing inter-bank copies right before their uses as in Figure 8 (b). In this approach, the secondary split live ranges will be shorter than in the first approach, reducing the register pressure of the secondary region. We do not have to worry about lengthening their primary split live ranges (thus increasing the register pressure of the primary region) because they were concurrently live at the region boundaries anyway, so lengthening them would not affect the register pressure of the primary region. And, even if lengthening primary split live ranges might cause new interferences with the region-local variables of the secondary region, they are supposed to be allocated from different banks, thus not competing for the same registers. Unfortunately, this approach might be disadvantageous in terms of code size if there are multiple uses (or multiple definitions for a live-out, region-across variable) because we need multiple inter-bank copies, while we add only one at the region boundary in the first approach. This might affect the code size negatively.

To compromise both approaches, we propose a new one. We first find all BBs in the secondary region that include the

[3] "]" means inclusion and ")" means exclusion.

uses. We then find a common *immediate dominator* of those BBs. Since the secondary region is an enclosing block, there must be such an immediate dominator inside the region. If the immediate dominator BB itself includes the use, we place the inter-bank copy just before the first use. Otherwise, we place the inter-bank copy at the bottom of the immediate dominator BB. By placing an inter-bank copy at the immediate dominator BB, we shorten the secondary split live range while generating only a single copy (similarly, for a region-across live-out variable, we place the inter-bank copy at the immediate post dominator BB of those BBs that includes its definitions). This is depicted in Figure 8 (c).

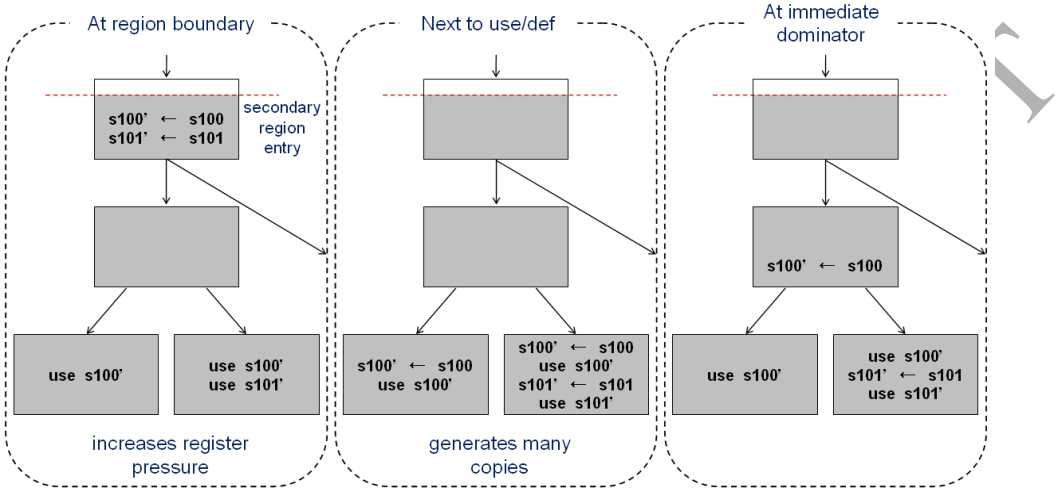


Figure 8. Three ways to put inter-bank copies for region-across variables

In Figure 9 (a), both `B2` and `B3` have region-across, live-in and live-out variable, `R0` (pre-colored). Their immediate dominator and immediate post dominator are `B1` and `B4`, respectively. Figure 9 (b) shows the result after the insertion of the inter-bank copies. We introduced two new pseudo registers `S16` and `S17` for splitting `R0` using inter-bank copies. Both are secondary split live ranges which will be allocated to the secondary bank registers. There is only one inter-bank copy merged with `mvb` for a region-across, live-in variable `R0` as `mvb S16, R0`, added at the bottom of B1. Similarly, we place an inter-bank copy for a region-across, live-out variable at the top of `B4`, as `mvb R0, S17`.

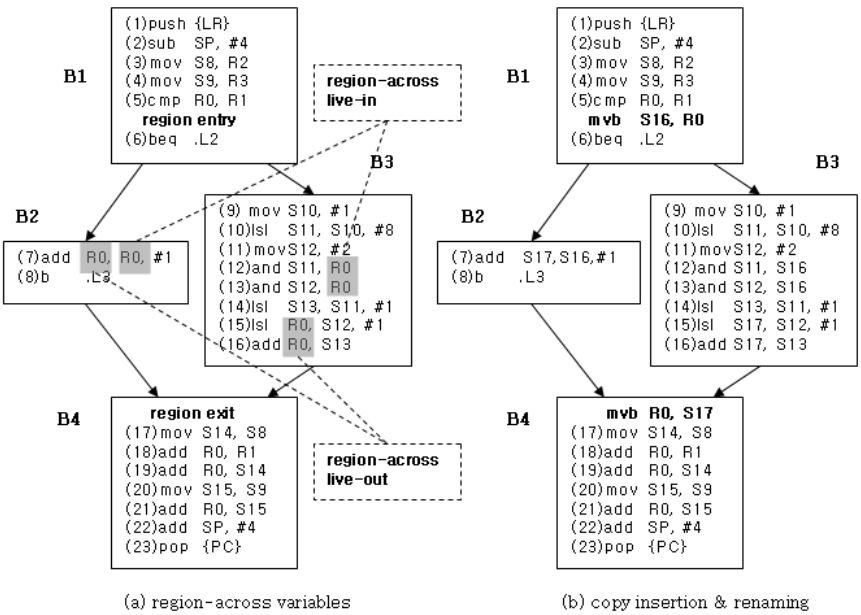


Figure 9. Insertion of bank change instructions and inter-bank copies

(7) Global Register Allocation: At this point, the code is partitioned into a set of primary regions and a set of secondary regions. Each variable in the code entirely belongs to either region and there is no more region-across variable. Therefore, we can perform global register allocation separately, one for the set of primary regions and one for the set of secondary regions. And as we mentioned, we simply allocate `R0~R3` to the primary regions and `R4~R7` to the secondary regions for better readability. The global register allocation process for each region is detailed in Figure 10.

```
function global_allocation(set of pseudo registers) {
  // STEP1: prepare for register allocation
  Precolor pseudo registers corresponding to arguments and return values;
  Do pseudo live analysis;
  Build a copy graph;

  // STEP2: group pseudo registers into LoRegs and HiRegs
  HiRegs := not-precolored pseudo registers that belong to the secondary region;
  LoRegs := not-precolored pseudo registers that belong to the primary region;

  // STEP3: allocate registers to the secondary region first
  Build an interference graph of HiRegs and update the copy graph;
  Make interferences between HiRegs and all machine registers of the primary bank;
  Give infinite spill cost to originally not spilled pseudo registers;
  SpilledHiRegs := color_graph(interference graph and copy graph of HiRegs);
  /* SpilledHiRegs is the set of those pseudo registers not colored; we give them a second chance */

  // STEP4: allocate registers to the primary region and do the actual spill
  LoRegs := LoRegs | SpilledHiRegs; // logical OR
  loop {
    Build an interference graph of LoRegs and update the copy graph;
    Make interferences between LoRegs and all machine registers of the secondary bank;
    Give infinite spill cost to originally not spilled pseudo registers;
    Give higher spill cost to LoRegs than SpilledHiRegs, so that LoRegs are colored first;
    SpilledRegs := color_graph(interference graph and copy graph of LoRegs);
    // if a SpilledHiRegs gets a color, its pseudo copies are replaced by inter-bank copies
    Break the loop if SpilledRegs is empty;
    actual_spill(SpilledRegs); // pseudo copies are replaced by memory spills
    Do pseudo live analysis;   // for building the graph again
  } // end of loop

  // STEP5: try to re-allocate memory spills in the primary region to the secondary bank
  Replace primary memory spills by pseudo copies, while recovering spill locations to pseudo registers;
  SpilledLoRegs := recovered pseudo registers;
  Do pseudo live analysis;
  Build an interference graph of SpilledLoRegs and update the copy graph;
  Make interferences between SpilledLoRegs and all machine registers of the primary bank;
  SpilledRegs := color_graph(interference graph and copy graph of SpilledLoRegs);
  // if a SpilledLoRegs gets a color, its pseudo copies are replaced by inter-bank copies
  actual_spill(SpilledRegs); // pseudo copies are replaced by memory spills
} // end of function
```

Figure 10. Register allocation details

The register allocation for both the primary region and the secondary region eventually calls the conventional graph-coloring algorithm **`color_graph()`** as shown in Figure 10, after the interference graph and the copy graph is built. In the current implementation, we used Briggs' optimistic coloring algorithm with iterated coalescing [4].

STEP1 is for the usual preparation of register allocation. STEP2 is for grouping pseudo registers into LoRegs and HiRegs, which are variables belonging to the primary region and the secondary region, respectively. We scan the code and

classify registers using the region boundaries. LoRegs and HiRegs will be allocated by `R0-R3` and `R4-R7`, respectively.

We first allocate registers to the secondary region in STEP3 (it does not really matter which region is allocated first, though). We build an interference graph and a copy graph for HiRegs. To prevent HiRegs from being allocated to primary bank registers (`R0-R3`), each variable in HiRegs has edges with all primary bank registers. We give an infinite spill cost to the pseudo registers which were not originally spilled in the THUMB code so that they are not spilled in the b-THUMB code either. We spill only those spilled in the THUMB code in a way of restoring the original spill code, which will be discussed shortly. The call to **`color_graph()`** returns those nodes that are failed to be allocated, *SpilledHiRegs*.

Instead of spilling SpilledHiRegs right away, we give them a second chance by adding them to LoRegs at STEP4, where we allocate registers to the primary region. Although we cannot color SpilledHiRegs directly with primary bank registers, we can use primary bank registers as spill locations using inter-bank copies, if there are otherwise unused primary bank registers available. This is similar to the copies to and from the invisible registers used in the THUMB code for register spills, and as such, it is not beneficial for code size but is advantageous for performance since we can avoid memory spills. Using inter-bank copies for SpilledHiRegs is permitted only when there are free primary bank registers left after allocating the primary region, though. This is implemented by adding higher spill cost to LoRegs than that of SpilledHiRegs.

The other process of STEP4 is similar to STEP3 except that we need to generate actual spills, if any, by making a call to **`actual_spill()`** in Figure 10. We make actual spills to those nodes unallocated in LoRegs and those not register-spilled in SpilledHiRegs. If there are actual spills, we repeat the allocation process, as in other graph-coloring techniques.

In STEP5, we try to allocate those nodes unallocated in LoRegs (which are called *SpilledLoRegs*) to the secondary bank registers, exactly as we tried to allocate SpilledHiRegs to the primary bank registers. We simply replace the memory spill instructions by pseudo copies, with the spill locations replaced by pseudo registers (which equals to SpilledLoRegs). Then, we build an interference graph for SpilledLoRegs and try to allocate them to the secondary bank registers available. If allocated, the pseudo copies become inter-bank copies; otherwise, they are restored back to the memory spill instructions.

We now explain how we generate the spill code. We make actual spills only to those nodes that are related to the originally-spilled code in a way of recovering the original spill code. We never make spills or generate spill instructions that are not in the original code. As we mentioned previously, the original THUMB code for a spilled variable `x` would be something like in Figure 11 (a) where we use store/load (or use a copy to and from an invisible register instead of store/load). The corresponding pseudo code after renaming would be as in Figure 11 (b) where there are three live ranges, `x`, `x'` and `s`. When we build an interference graph, `x`, `x'` and `s` will be nodes in the graph. Among these three nodes, we give the minimum spill cost to **`s`**, so that **`s`** should be spilled if needed, while neither `x` nor `x'` is spilled. If we do spill `s`, we simply convert the copies that contains `s` into a store (`copy s=x` → `store x @spill`) and a load (`copy x'=s` → `load x'@spill`) as shown in Figure 11 (c), which restores the original spill code. If `s` were in SpilledHiRegs after we allocate the secondary region and if there is a primary bank register available for it, we generate inter-bank copies instead of store/load, as in Figure 11 (d) (we do the same for SpilledLoRegs to make an inter-bank copy to the secondary bank). If there are no spills in Figure 11 (b), it is likely that both `copy s=x` and `copy x'=s` are coalesced, so that `x`, `x'`, `s` are allocated to the same register.

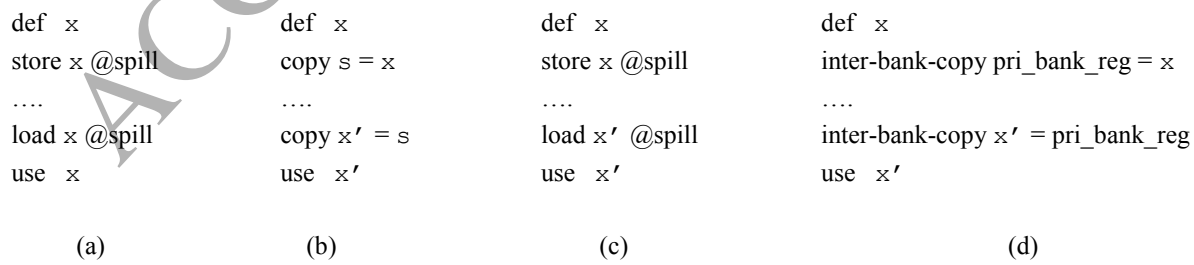


Figure 11. Handling of Actual Spills for Originally-Spilled Nodes

Back to our example in Figure 9 (b), the previous spill instructions (3), (4), (17), and (20) are going to be removed by coalescing and register allocation this time. In addition, the stack adjustment instructions (2) and (22) will no longer be needed. After the global register allocation, we perform a peephole optimization using code motion to merge bank change instructions with (inter-bank) copies that were not possible when making interfaces for the secondary region. The code motion is allowed only when it does not break data dependences. The final result code will be as in Figure 5 (c).

5 Experimental Results

In this section, we evaluate the proposed banked register model and the banked register allocation technique on the target b-THUMB architecture. We first describe the experimental environment. Then, we present the code size and the performance results, with other relevant data useful for understanding the results.

5.1 Experimental Environment

We performed our experiments using a modified THUMB-based GNU tool chain. Our region-based banked register allocator works as a post-pass assembly optimizer, which takes a THUMB assembly generated by GCC as input and produces a b-THUMB assembly. In this way, we do not have to develop a dedicated compiler for the b-THUMB architecture but can get the b-THUMB assemblies from the existing ones, for fast prototyping. Moreover, we can isolate the benefit of our banked register allocation from that of GCC's register allocation when analyzing the result of spill reduction, which will be helpful for evaluating our banked register allocation separately. However, if we could integrate banked register allocation with other optimizations (e.g., instruction scheduling), we might be able to generate better code.

In the THUMB assembly code generated by GCC, we added three annotations to each function, the *frame size*, the *outgoing argument size*, and the *subscripted variable size*, for detecting spill instructions. GCC allocates subscripted variables, such as arrays or variables referenced by pointers, to the bottom of the stack frame, while the space for argument passing is allocated to the top of the stack frame. The space for spills is located between them, so we can identify memory spills by checking load/store instructions whose stack offset is between "*outgoing argument size*" and "*frame size*" – "*subscripted variable size*". We also identify spills to invisible registers. These spills are converted into copies to pseudo registers as we explained. We then allocate banked registers and generate the b-THUMB assembly code. If the size of the b-THUMB assembly code is bigger than the original THUMB assembly code for a given function (it rarely occurs, though), we give up the b-THUMB assembly and use the THUMB assembly instead (this is fine since the b-THUMB is mostly backward compatible with the THUMB such that the THUMB code can run on the b-THUMB machine).

The b-THUMB assembly is fed to a modified GAS assembler that understands the b-THUMB ISA, and to a general THUMB ELF linker. We modified the ARM simulator embedded in GDB to run our b-THUMB binary which produces results and statistics. We do not include the standard libraries into the binary, but include only the dummy functions which have a software-interrupt instruction. When a library function is called, the simulator catches the software interrupt. Then an appropriate interrupt service routine collects arguments from the simulator's internal data structure and run a host library function with the arguments. The result is written to the simulator's data structure representing the b-THUMB registers. As the original GDB simulator, it does not perform cache simulation.

Our benchmarks are composed of six programs from MediaBench [1] and MiBench [2], which are well-known benchmarks for embedded systems. They are *adpcm* (Intel/DVI ADPCM codec), *basicmath* (mathematical tests for automotive area), *gsm* (speech compression), *g721* (CCITT G.711, G721, and G723 voice compressions), *jpeg* (image compression and decompression), and *mpeg2dec* (moving pictures and associated audio de-compressor). Other programs are not included either because they are too small to generate any spills or because they use complex libraries which we could not make work correctly due to a large amount of assembly coding required (we successfully re-implemented floating point math library with the b-THUMB code, though) or a more elaborate argument passing mechanism than the simple one described above (most of memory managing libraries are supported by our own memory abstraction, though).

5.2 Spill Ratio in the ARM Code and in the THUMB Code.

Figure 12 shows the portion of spill instructions in the ARM assembly code (left) and in the THUMB assembly code (right), respectively. They are generated by GCC 3.0 with a code size priority option *-Os*. The ARM code includes only spills to the memory using load/store (depicted by *memory spill*), while the THUMB code also includes spills to invisible registers (`R8-R12`) using copies (depicted by *register spill*). Although the THUMB code is known to be more compact, 12.5% of its instructions are used for spills, whereas only 4.2% of instructions are used for spills in the ARM code. The performance degradation due to spills in the THUMB code would not be serious since much of them are register spills, though, as can be seen in the graph. Nonetheless, the graph shows that reduced instruction encoding in THUMB leads to more spills than ARM, due to the shortage of registers. In fact, the spill ratio with our small benchmarks is not that high even for THUMB, but with real, larger embedded applications, the ratio is likely to be much higher, causing more serious spill overhead and code expansion. Therefore, our attempt to reduce spills using banked registers is justified.

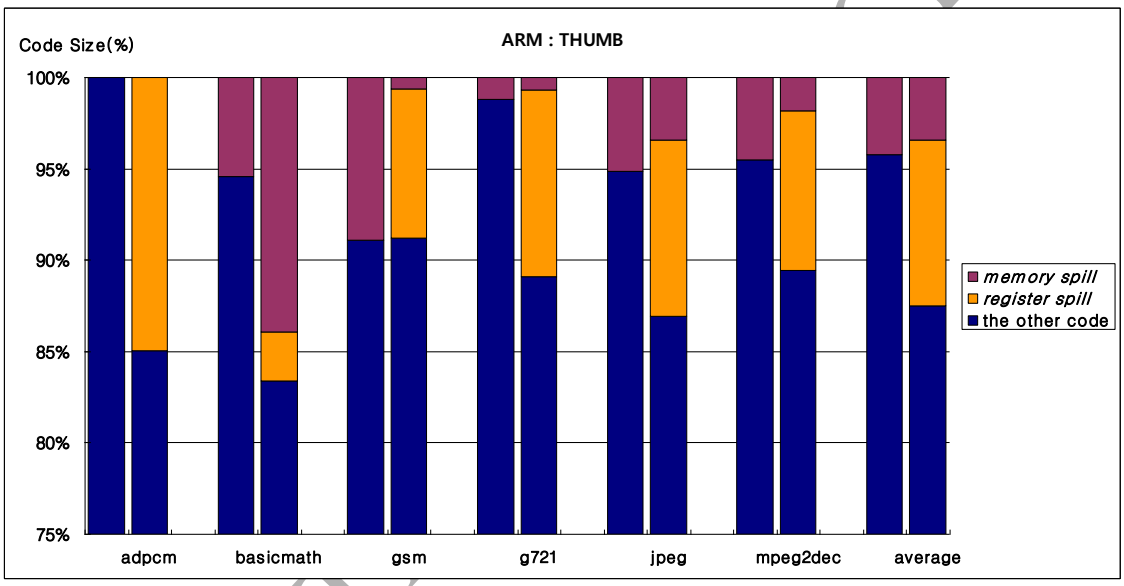


Figure 12. Portion of spills in THUMB and ARM

5.3 Code Size Reduction

Figure 13 depicts the code size ratio of the b-THUMB code (right), compared to the original THUMB code (left) as 100% for each benchmark. It shows that b-THUMB reduces the code size by more than 5.8% on average (8% in peak). Both bars include *memory spill* and *register spill*, while the right bar (b-THUMB) also includes the portion of inter-bank copies and bank change instructions not merged with copies. We can see that the code size of b-THUMB is decreased mainly due to reduction of spill code, while the overhead of bank changes (i.e., inter-bank copies and bank changes) is small (2.1%). Table 3 gives some of the raw data of Figure 13, which also show the total number of functions and the number of functions that spill for each benchmark.

The *register spill* in b-THUMB mostly means those inter-bank copies for SpilledHiRegs and SpilledLoRegs. It is aimed at increasing the performance, not reducing the code size, as explained previously. Except for *basicmath*, *memory spill* is increased a little in most benchmarks which will be discussed shortly.

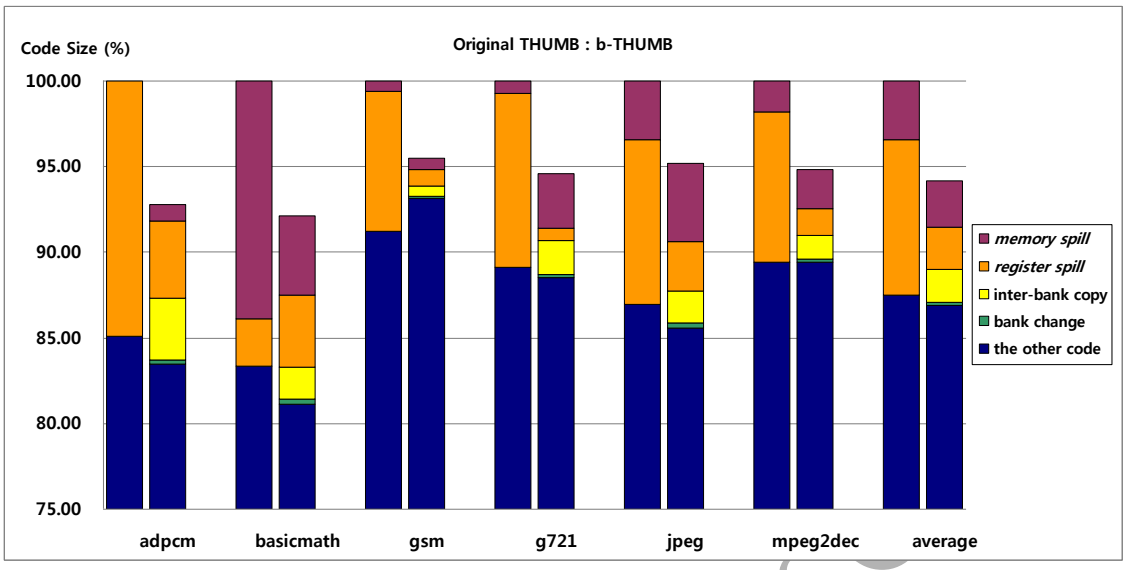


Figure 13. Code size and spill reduction

Table 3. Raw data for Figure 13

Benchmark	# of functions	# of functions with spills (ratio)	Original code size	Original spills	New code size	Remaining spills	Inter-bank copies	Bank changes
adpcm	5	3 (60%)	442	66	410	24	16	1
basicmath	5	2 (40%)	992	165	914	88	18	3
gsm	94	26 (27%)	10461	918	9988	170	64	10
g721	28	12 (43%)	2156	235	2040	85	42	4
jpeg	463	195 (42%)	42582	5556	40529	3162	811	127
mpeg2dec	114	39 (34%)	11006	1163	10440	429	150	22

We also measured the code size reduction ratio separately for those functions that originally spill in Table 3, since functions that do not spill cannot benefit from banked register allocation. In this case, the average code size reduction ratio goes up to 7.6% as shown in Figure 14. These results indicate that our approach reduces the code size tangibly.

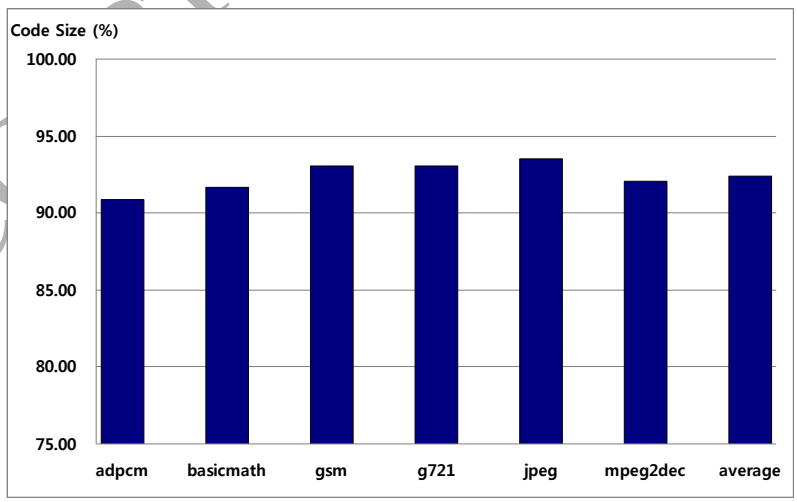


Figure 14. Code size reduction as to the code of functions with spills

5.4 Performance Evaluation

Although our main goal is reducing the code size, we also check the performance impact of banked register allocation by measuring the cycle count during simulation. This evaluation is just from the compiler perspectives, not considering any micro-architectural complexity, so it should not be interpreted as the performance of the banked register architecture. In fact, our GDB-based simulator cannot simulate the cache behavior, so we performed two simplified experiments, one with a perfect cache and one with no cache to see the performance boundary only. For the perfect cache experiment, we assume that the load takes two cycles, while all other instructions take a single cycle, as in most RISC processors. For the cache-less experiment, we assumed that loads and stores take four cycles, while others take a single cycle[4].

Figure 15 shows the result of the perfect cache experiment. It shows the ratio of cycle counts of the b-THUMB (right) compared to those of the THUMB (left) as 100%. The b-THUMB outperforms the THUMB for four benchmarks and shows a similar result for the other two benchmarks, achieving an average of 3.3% and a peak of 11.1% performance improvement. Generally, smaller code size due to the reduction of spills would lead to better performance if experimented on the same environment, but there is a big difference between the THUMB environment and the b-THUMB environment, in terms of spills. That is, the THUMB is already using the invisible registers as spill locations thru inter-bank copies, to avoid performance degradation caused by memory spills. In fact, Figure 13 showed that most spills in the THUMB are register spills (except for *basicmath*). On the other hand, although we can reduce the absolute number of spills (hence spill instructions) using banked registers in the b-THUMB, most of spills in the b-THUMB are now memory spills since both banks are now being used actively (inter-bank copies for SpilledHiRegs and SpilledLoRegs are our attempt to reduce memory spills and if we do not do this, the performance is much worse, as we will see shortly). This result indicates that using the invisible registers for register allocation using the register bank is better than using them as spill locations.

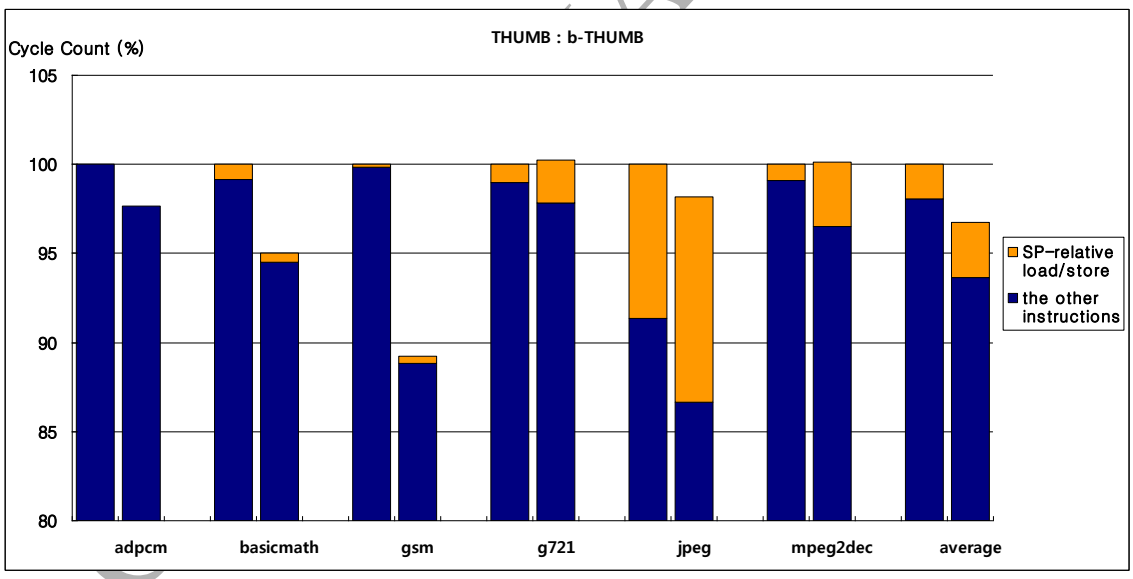


Figure 15. Execution time (cycle count) with a perfect cache

At the top of each bar in Figure 15, the portion of loads and stores that access SP-relative addresses is included. Not all of the SP-relative memory accesses are spills, but both the THUMB and the b-THUMB have the same amount of non-spill

[4] The cycle count is the sum of three types of ARM cycles, i.e. S-cycle (sequential cycle), N-cycle (non-sequential cycle), and I-cycle (internal cycle). In summary, the usual data processing instructions consume 1S cycle, the load instruction consumes 1S+1N+1I cycles, and the store instruction consumes 2N cycles [3]. In a low-end cache-less ARM embedded system, N-cycle typically takes twice longer than the other cycles [27], so we need to count double for N-cycles. This makes memory instructions take four cycles.

SP-relative accesses, so the difference can be interpreted as the count of additional memory spills executed in the b-THUMB. The difference of dynamic count of memory spills is more or less consistent with the static count of memory spills in Figure 13, except for *basicmath* and *adpcm*, where it appears that memory spills are not in frequently-executed paths. Most importantly, Figure 15 shows that for those two benchmarks where the b-THUMB does not improve the performance of THUMB (*g721*, *mpeg2dec*), a substantial amount of additional memory spills are executed, while there is a little difference for the other four benchmarks where the b-THUMB outperforms. This is consistent with the performance differences, explaining the reason.

We also evaluate the performance impact of those inter-bank copies for SpilledHiRegs and SpilledLoRegs. Figure 16 depicts the cycle count ratio of the b-THUMB when those inter-bank copies are replaced by memory spills, compared to the b-THUMB. On average, the cycle count increases by 1.2%, yet it increases by 5.3% in *adpcm*. This result indicates that those inter-bank copies are important for improving the performance, although there is no impact on the code size.

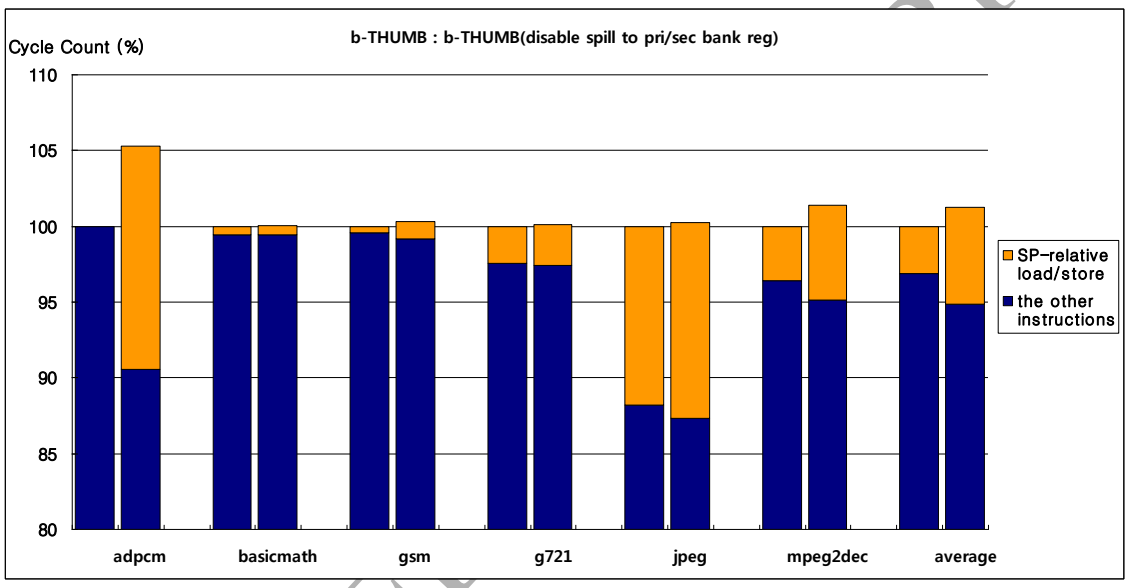


Figure 16. Performance loss without spills to primary/secondary bank registers

It might be questioned if the separation of the three special registers in b-THUMB leads to an unfair comparison to THUMB since it means that b-THUMB has three more GPRs than Thumb. From the code size perspective, however, the comparison is fair. Even if we added three more GPRs to Thumb, they would be used only for register spills because they are invisible registers. This means that the best optimization with those three registers would be replacing memory spills, if any, by register spills. In this case, the number of instructions remains the same, not affecting the code size. From the performance perspective, there might be an improvement since register spills are more efficient than memory spills. However, Figure 15 implies that the original THUMB executes a small amount of SP-relative memory instructions (some of them are memory spills), so replacing memory spills by register spills would not improve THUMB's performance much.

Figure 17 shows the result of the cache-less experiment. Since memory spills take four times longer cycles than register spills in our environment, the b-THUMB is seriously disadvantaged. Nonetheless, the performance of the b-THUMB is still competitive to that of the THUMB, indicating that the b-THUMB is also useful in cache-less embedded systems.

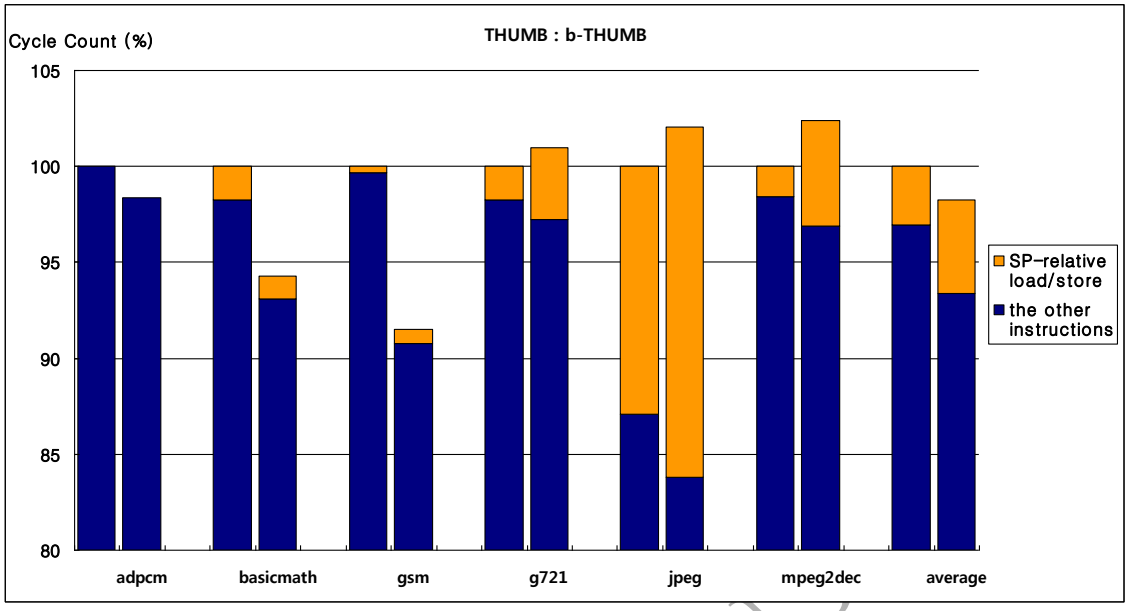


Figure 17. Execution time (cycle count) without a cache

5.5 Evaluation of the Different Insertion Policies of Inter-bank Copies

Our b-THUMB uses the immediate dominator relationship to determine the location of inter-bank copies as explained in Section 4 (6). In Figure 18, we compare the code size of our approach (second bar) with the two others: placing all inter-bank copies at the region boundaries (third bar) and placing inter-bank copies right next to their definitions and uses (fourth bar). The *copy-at-boundaries* shows a relatively higher spill portion in the bar due to its increased register pressure, increasing the code size than ours, while the *next-to-use/def* tends to increase the other portion due to duplicated copies, increasing the code size further. This result shows that the dominator-based location is superior.

5.6 Comparing b-THUMB to THUMB with Additional Registers

To evaluate how well our banked register allocation reduced the code size, we compare with a hypothetical THUMB that is assumed to have more registers than the original 8 registers. We first experimented with a THUMB that is assumed to use all of the 13 GPRs of the ARM processor (expressed as *13GPRS*). We also experimented with a THUMB that is assumed to have 16 GPRs such that the three special registers (`SP`, `LR`, `PC`) are separately located as in b-THUMB (expressed as *16GPRS*). It should be noted that these hypothetical THUMBs use the additional registers for register allocation, not as spill locations. We modified GCC's code generator and machine description to generate the hypothetical THUMB code. Figure 19 compares the code sizes of the original THUMB, b-THUMB, *13GPRS*, and *16GPRS*.

As we can expect, *16GPRS* and *13GPRS* lead to shorter code sizes with less spills than the original THUMB, since there are more registers available for register allocation. Comparing *16GPRS* and the b-THUMB would be interesting since both use 16 registers, yet *16GPRS* use 16 registers fully, while the b-THUMB use 16 registers in a banked form. Figure 19 shows that *16GPRS* produces shorter code than b-THUMB by 3.4% on average, yet considering the 2.1% overhead for bank changes in the b-THUMB, we believe that our register allocator did a good job of assigning banked registers.

Although most graphs typically show that *16GPRS* is the best while *13GPRS* is the next, *basicmath* shows that both *16GPRS* and *13GPRS* lead to a worse code size than the b-THUMB. We found that *gcc* generates unnecessary copies for using the additional registers (for a constant, floating-point argument, they first load it to a general register then copy it to an argument register, while it can be loaded directly to the argument register in the original THUMB).

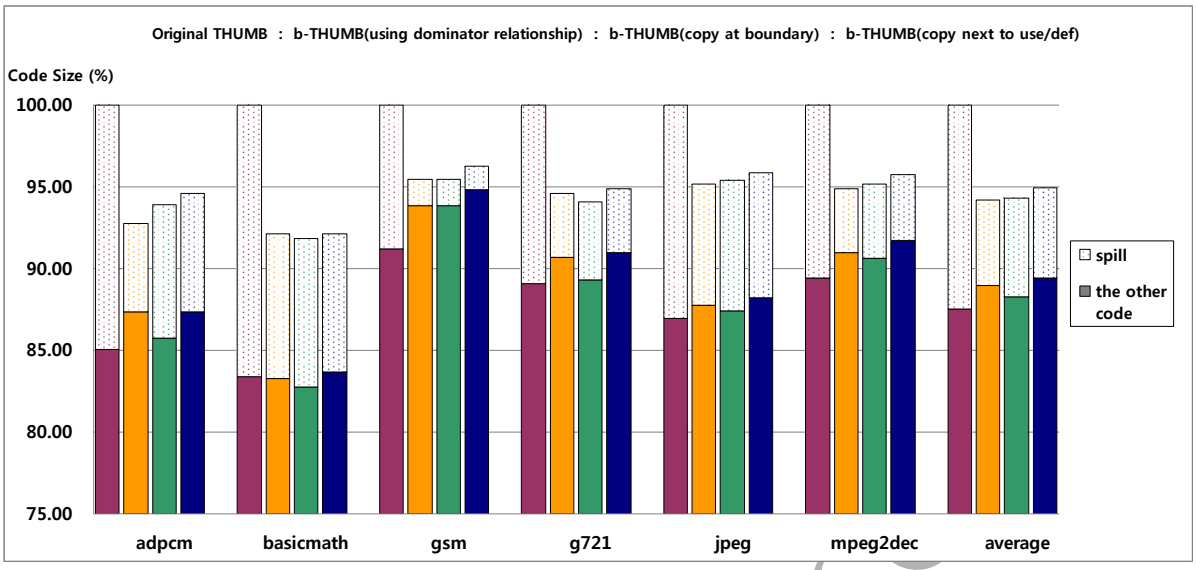


Figure 18. Code size of different insertion policies of inter-bank copies

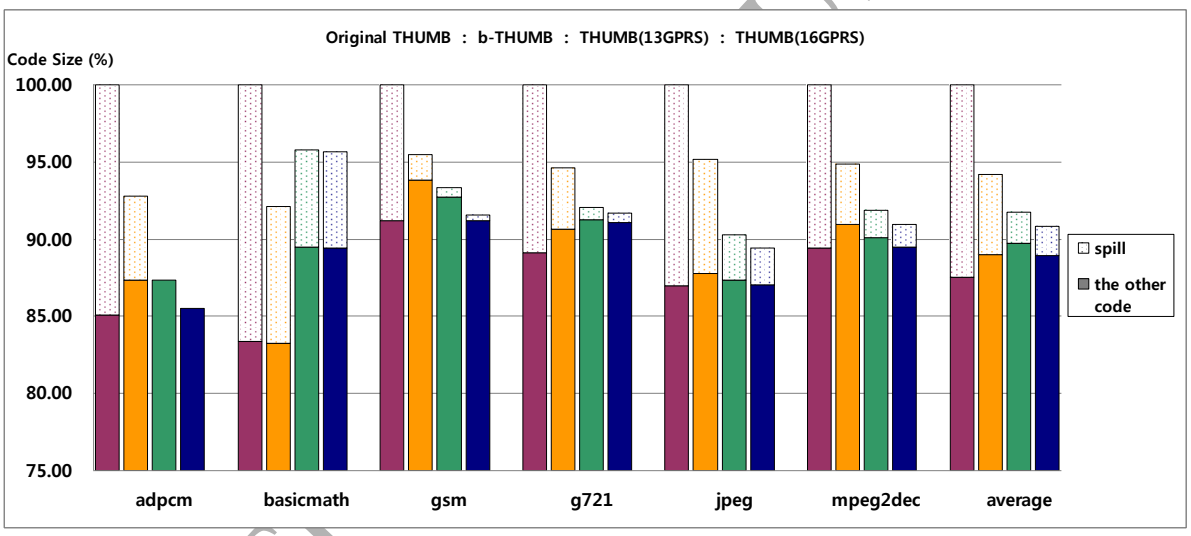


Figure 19. Comparing the b-THUMB to the THUMBs with additional registers

5.7 Code Size Comparison with Fewer Registers

We evaluate banked register allocation when the register pressure is higher. We reduce the number of available registers in the original THUMB, which is 8 GPRs, to 7 GPRs, 6 GPRs, and 5 GPRs, and compare their code sizes with those of the corresponding b-THUMBs, which have 14 GPRs, 12 GPRs, and 10 GPRs, respectively, in a banked form. As previously, the THUMB uses the invisible registers as spill locations. We generate the THUMB code by modifying the GCC's code generator, and we change the calling convention appropriately since fewer registers are available.

Figure 20 shows the code size ratio of the b-THUMBs compared to the original THUMB as 100% for each register configuration. We can find that the b-THUMB code size is consistently smaller than the THUMB code size as in the 8 GPRs case, and the ratio gets smaller as the number of registers decreases, meaning that higher register pressure leads the b-THUMB to produce even shorter code than the THUMB (the average difference between 8GPRs and 5GPRs is 1.8%).

In summary, our results show that b-THUMB can decrease the code size and improve the performance tangibly.

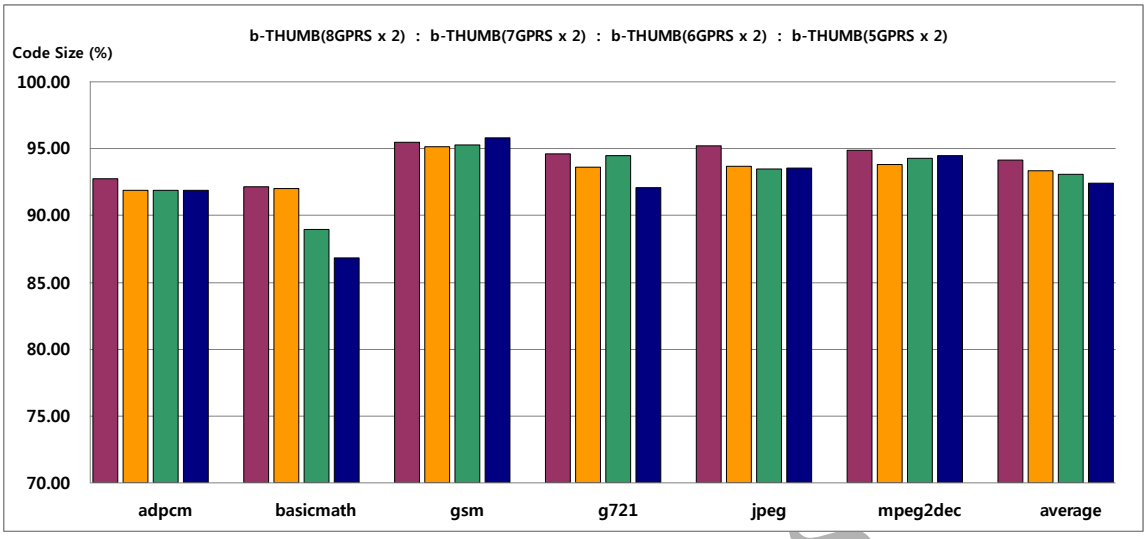


Figure 20. Comparing b-THUMB with THUMB (100%) when there are fewer registers

6 Related Work

Banked registers have been used for various purposes in diverse contexts. We first describe previous work that employs banked registers for short encoding, thus directly comparable to ours. We then describe related work in different contexts.

6.1 Banked Registers for Short Encoding Architectures

An interesting modification to the ARM THUMB architecture is proposed for utilizing its invisible registers [16]. It provides a `SetMask` instruction which includes an 8-bit bit mask as an operand, representing the visible 8 registers. If some bit of the mask is set, the corresponding register number used thereafter actually refers to the matching one in the invisible registers. For example, `SetMask 00000001` makes `R0` (corresponding to the rightmost bit) refer to `R8`. This allows allocating all of the original 16 ARM registers without any restriction because whenever a THUMB instruction has operands allocated to invisible registers, a `SetMask` is added right before the instruction that sets the corresponding bits and another `SetMask` is added right after it that resets them. To reduce the number of `SetMask`s, optimization is performed to merge them as much as possible. They also claim that the performance overhead caused by `SetMask`s can be reduced by achieving zero-cycle `SetMask` using dynamic instruction coalescing [15]. They report an average of 5% performance improvement even with the zero-cycle `SetMask`, which is not much higher than ours. Unlike our work, they failed reducing the code size, though, even when they perform optimizations to remove as many `SetMask`s as possible.

Although this `SetMask`-based partitioning can allow more flexible regions (possibly beyond basic blocks), which use registers selectively from both banks and obviate inter-bank copies, its code size essentially suffers from too many `SetMask`s, due to its register allocation irrespective of banks; in fact, register allocation considering banks does not appear straightforward, either. The performance could also be affected seriously if there is no hardware coalescing support.

Register window is proposed to increase the number of available registers for compact encoding architectures without compromising its instruction encoding [20]. Register window is similarly motivated to our register bank, so they deal with the same banked (windowed) register allocation problem. They employ bank selection instead of bank toggling so that its bank change instruction explicitly specifies the bank number to be active, which allows them to have more than two banks. Register partitioning, similar to our region partitioning, is performed to decide which pseudo registers are allocated to

which register banks. Region partitioning is performed on a unit of basic block (BB) or a superblock (their examples and algorithms are based on BBs, though) such that the region entry always starts with bank 0 and the pseudo registers within the region is partitioned into register banks. They achieve an average of 10% performance improvement with dual banks (i.e., by doubling up the number of available registers) on a WIMS microcontroller where every instruction takes a single cycle. We cannot say that this result is better than ours since our performance improvement would increase to an average of 6.4% if we also assume single-cycle instructions. Moreover, our base machine (THUMB) is already using the invisible registers as spill locations, so it is hard to improve performance by reorganizing those registers into banks only, while their base machine is not aware of those new registers (i.e., they are essentially doubling up the available registers while we are not). As to the code size, they do not report any result since their primary goal is improving the performance and reducing the power consumption, not reducing the code size, unlike our work.

There might be redundant bank changes caused by those added at each region entry to start the region with bank0; many of them are unnecessary if the predecessor regions also end with bank0, and their optimization removes some of the obvious ones but not all. Moreover, bank selection requires a function call to be made only at bank0 which may require additional code for transferring arguments or return values as well as bank changes. The selection instructions cannot be merged with copies unlike bank toggling. Therefore, this approach is likely to be suffered from too many bank changes.

We also tried with bank selection instead of bank toggling in our b-THUMB architecture [37]. Bank selection, in theory, could lead to more flexible register allocation (it can even allow non-enclosing block in Figure 7 (b)), leading to better distribution of register pressure. Unfortunately, our evaluation indicates that bank selection even with an aggressive hardware support does not show any code size or performance benefit compared to bank toggling although it can allow a better partition of regions, due to excessive inter-bank copies generated [37].

Our previous work for Samsung's CalmRISC8 microcontroller also deals with the banked register allocation problem [19]. CalmRISC8 has four register banks, yet only two banks are visible in the user mode [21]. Its bank change instruction is based on bank selection, not bank toggling. Its compiler performs a simple local register allocation with left-edge algorithm at the secondary bank, followed by a global register allocation at the primary bank. Moreover, the secondary bank region cannot exceed a BB. Our current work overcomes these limitations and provides a more elaborate technique.

6.2 Banked Registers for Other Purposes

Intel MCS51 processor family is a well-known microcontroller that has banked registers [9]. It has a total of 32 registers but they are divided into four banks, 8 registers each. Microcontrollers often deal with many interrupt requests, and register banks can be used effectively for fast interrupt handling and context switching. That is, if we use different banks for different interrupt service routines, there is no need for register saves and restores, improving the performance. A similar idea is exploited in Zilog Z8 processor [26], which has 16x16 banked registers with one active bank at a time, but can also access the whole 256 registers using 8-bit addressing. A dedicated register bank is often allocated to an interrupt handler as in MCS51. There is a research work for code generation of Z8 based on integer linear programming (ILP) [18] with an emphasis on interrupt handlers, yet it does not specifically deal with banked register allocation.

There are usages of banked registers for energy saving by reducing the size of register ports [5][6][8][14]. Ayala et al. shows that dividing a register file into smaller groups (banks) is beneficial in reducing the power consumption due to smaller ports and faster access time [5]. Register allocation for such architectures attempt to assign values into different banks as equally as possible; even the operands of an instruction are preferred to being allocated to different banks. Hiser et al. deal with a similar register allocation problem [8]. Cruz et al. exploit multi-level structure of register banks [6], while Kondo et al. propose a hardware that selects the register bank automatically in microcode level [14].

Another energy saving technique using banked registers is based on an observation that there are hot registers accessed frequently during execution [36]. By separating hot registers and cold registers ones into different banks and by reallocating registers in the code, the proposed technique could reduce power consumption by more than 50%.

Intel IXP network processor family uses a dual-bank register file for fast, parallel data fetches. This architecture has a constraint that the ALU source operands should come from different banks yet there is no such a constraint for the target

register. Zhuang et al. [24] and Zhou et al. [23] discuss how to solve register conflicts occurred from the constraints.

Besides the banked register files, there have been several attempts to solve the shortage of registers or memory by providing some extended hardware resources. The following researches correspond to such efforts.

Register Connection was quite a novel idea at the time of publication [13], which is adding a set of extended registers into the CPU to overcome the shortage of registers. They introduced register mapping table and register connection instruction to use the extended registers. It is similar to the approach of the PDP-11, which maps a small addressable memory space into a larger physical memory. Cooper et al. proposed an approach similar to this extension [11]. Recently, Dressen et al. proposed mapping a block of architected registers to multiple blocks of physical registers so as to reduce the memory accesses. As others, their focus is improving performance even if the code size increases, which contrasts to ours.

Zhuang et al. made a similar observation to ours that for some architectures the number of registers exposed to the programmer is much smaller than the number of physical registers, because of backward compatibility or encoding space of register fields [25]. The DSP56300, StarCore SC110, ARM THUMB and MIPS-16 are such examples. To allocate more physical registers, they propose a hardware managed register allocation scheme where the compiler gives a hint of allocation priority via spill offset numbers in the code, and then the hardware identifies such spills and tries to allocate them to a separate hardware-managed register file at runtime.

Scholz et al. [22] proposed memory bank switching to increase the size of the code and data memory without extending the address bus of a CPU. The address space is partitioned into memory banks, and the CPU can only access one bank at a time. They attempt to minimize the number of bank selection instructions for a given code to reduce their overhead.

Clustered VLIW architectures have been proposed for better scalability of the VLIW performance [32]. In such processors, functional units are partitioned into clusters so is the register file into register banks. Therefore, banked register allocation is also important for reducing inter-bank communication, yet it should be closely integrated with instruction scheduling and resource allocation [33,34,35], rather than working on its own.

6.3 Other Approaches to Code Size Reduction for Reduced Instruction Encoding Architectures

Unlike our approach to code size reduction based on banked registers, there have been other approaches to code size reduction for reduced instruction encoding architectures.

Shrivastava et. al proposed a compilation framework for code size reduction by generating instructions for a reduced ISA (rISA), mixed with the instructions for its normal ISA (applicable to ARM/THUMB or MIPS-32/MIPS-16) [29]. They found that the rISA code is not always smaller than the ISA code because of additional instructions needed to perform the same computation, unavailability of compound instructions in rISA such as prediction or speculation, and register spills due to shortened register fields. Unlike other compilers that generate either the rISA code or the ISA code for a given routine depending on its rISA profitability, the proposed framework can generate code mixed with both, depending on the rISA profitability of different regions within the routine. They found that this leads to better code size reduction. Unlike our work, however, they do not deal with register allocation issues seriously.

Kwon et. al proposed redesigning the THUMB ISA by allocating one bit to compress more instructions [30] or to specify parallel execution of a pair of instructions [31]. The one bit is stolen from the destination register field, which leads to partitioning the instructions into two groups, one that can access R0~R3, and the other that can access R4~R7 as a destination register. For this heterogeneous register set, they employ an elaborate register allocation technique to avoid unbalanced register usage, hence additional copies or spills. We have a similarly-motivated issue of balancing the register allocation for the primary and the secondary bank, yet each bank is homogeneous, so the allocation problem is different.

7 Summary and Future Work

Code size is important in many embedded systems for reducing memory cost, power consumption, and I-cache pressure. Reduced encoding architecture is one popular hardware solution to achieve small code size. Unfortunately, reduced

encoding for register operand fields makes fewer registers available for register allocation, leading to more spills and affecting the code size and the performance negatively, although those invisible registers can be used as spill locations to mitigate the performance penalty as in the THUMB code. This paper proposed reorganizing its register file into a banked one so that all of the original registers are available for register allocation to improve the code size. This is not straightforward because bank changes and inter-bank copies cause an overhead, and the invisible registers are already used as spill locations. So, we used an elaborate banked register allocation technique, which partitions the code into primary and secondary regions, adds bank toggles and inter-bank copies, and allocates them separately. We can spill only a subset of those spilled in the original THUMB code in a way of restoring the original spill code. We prefer register spills based on inter-bank copies to memory spills to reduce the performance overhead.

Our experimental results show that the banked register model for THUMB (b-THUMB) leads to a 5.8% reduction of code size and a competitive (3.3%) performance improvement. This result should not be interpreted as a precise micro-architectural evaluation of the b-THUMB compared to the THUMB (or THUMB2), but an evaluation of the banked register model for reduced encoding architectures.

Generally, when one wants to build a reduced encoding version of a given CPU for embedded systems, there will be a choice of using the unavailable registers as spill locations or reorganizing them as a separate bank. Our results indicate that the latter is better for both the code size and the performance. When one wants to double the number of available registers for a given compact encoding CPU while keeping its short instruction encoding, our proposal for banked registers and region-based banked register allocation would also be useful for generating efficient code for it.

If we have more than two banks, we would need a bank change mechanism other than the bank toggle, a key component of the proposed technique that simplifies code generation and banked register allocation. We would also need a different region partitioning algorithm, a different insertion heuristics for inter-bank copies, and a new way of assigning regions to banks, which would lead to more elaborate banked register allocation technique than ours. This is left as a future work.

References

[1] MediaBench. http://euler.slu.edu/~fritts/mediabench/mb1/

[2] MiBench. http://www.eecs.umich.edu/mibench/

[3] Advanced RISC Machines Ltd. *ARM7TDMI Technical Reference Manual*, rev 4 edition, Sep 2001.

[4] P. Briggs, K. D. Cooper, and L. Torczon. Improvements to Graph Coloring Register Allocation. *ACM Transactions on Programming Languages and Systems*, pages 428-455, May 1994.

[5] J. L. Ayala, M. Lopez-Vallejo, and A. Veidenbaum. A Compiler-assisted Banked Register File Architecture. In *Workshop on Application Specific Processors* , Sep 2004.

[6] J.-L. Cruz, A. Gonzalez, M. Valero, and N. Topham. Multiple-Banked Register File Architectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 316–325, Jun 2000.

[7] ARM. Improving ARM Code Density and Performance. http://www.arm.com/pdfs/Thumb2CoreTechnologyWhitePaper-Final4.pdf

[8] J. Hiser, S. Carr, and P. Sweany. Register Assignment for Software Pipelining with Partitioned Register Banks. In *International Parallel and Distributed Processing Symposium*, 2000.

[9] Intel Corporation. *MCS51 Microcontroller Family User's Manual*, Feb 1994.

[10] R. Johnson, D. Pearson, and K. Pingali. Finding Regions Fast: Single Entry Single Exit and Control Regions in Linear Time. Technical report, Cornell University, 1993.

[11] K. D. Cooper and T. J. Harvey. Compiler-Controlled Memory. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

[12] K. Kissell. *MIPS16: High-Density MIPS for the Embedded Market*. Silicon Graphics MIPS Group, 1997.

[13] T. Kiyohara et al. Register Connection: A New Approach to Adding Registers into Instruction Set Architectures. In *20th Annual International Symposium on Computer Architecture*, 1993.

[14] M. Kondo and H. Nakamura. A Small, Fast and Low-Power Register File by Bit-Partitioning. In *11th International Conference on High-Performance Computer Architecture*, Feb 2005.

[15] A. Krishnaswamy and R. Gupta. Dynamic Coalescing for 16-Bit Instructions. *ACM Transactions on Embedded Computing Systems(TECS),* Feb 2005.

[16] A. Krishnaswamy and R. Gupta. Efficient Use of Invisible Registers in Thumb Code. In *Proceedings of the 38th IEEE/ACM Internal Symposium on Microarchitecture*, Nov 2005.

[17] J.-H. Lee, J. Park, and S.-M. Moon. Securing More Registers with Reduced Instruction Encoding Architectures. In *Proceedings of 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, Aug 2007.

[18] M. Naik and J. Palsberg. Compiling with Code-size Constraints. *ACM Transactions on Embedded Computing Systems*, 3(1):163–181, Feb 2004.

[19] J. Park, J.-H. Lee, and S.-M. Moon. Register Allocation for Banked Register File. In *Proceedings of the ACM SIGPLAN workshop on Languages, Compilers and Tools for Embedded Systems*, Aug 2001.

[20] R. A. Ravindran et al. Partitioning Variables across Register Windows to Reduce Spill Code in a Low-Power Processor. *IEEE Transactions on Computers*, 54(8), Aug 2005.

[21] Samsung Electronics Corporation. *CalmRISC8 Specification*, 1999.

[22] B. Scholz, et al. Minimizing Bank Selection Instructions for Partitioned Memory Architectures. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, 2006.

[23] F. Zhou et al. A Register Allocation Framework for Banked Register Files with Access Constraints. In *Asia-Pacific Computer Systems Architecture Conference*, 2005.

[24] X. Zhuang and S. Pande. Resolving Register Bank Conflict s for a Network Processor. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, 2003.

[25] X. Zhuang, T. Zhang, and S. Pande. Hardware-managed Register Allocation for Embedded Processors. In *Proceedings of the ACM SIGPLAN conference on Languages, Compilers and Tools for Embedded Systems*, 2004.

[26] Zilog. *Z8 Microcontroller User's Manual.*

[27] Advanced RISC Machines Ltd. *Application Note 26 "Benchmarking, Performance Analysis and Profiling"*, 1995.

[28] Motorola, *CPU12 Reference Manual*, http://www.freescale.com/files/microcontrollers/doc/ref_manual/CPU12RM.pdf, Jun 2003.

[29] A. Shrivastava, P. Biswas, A. Halambi, N. Dutt, A. Nicolau. Compilation Framework for Code Size Reduction using Reduced bit-width ISAs, *ACM Translations on Design Automation and Electronic Systems*, 11(1), 2006.

[30] Y.-J. Kwon, X. Ma, and H. Lee. PARE. Instruction Set Architecture for Efficient Code Size Reduction. *Electronics Letter*, Vol. 35 No. 24, 1999.

[31] Y.-J. Kwon, D. Parker, and H. Lee. Toe. Instruction Set Architecture for Code Size Reduction and Two Operations Execution. In *Proceedings of the 1999 International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES)*, 1999.

[32] A. S. Terechko and H. Corporaal. Inter-cluster communication in VLIW architectures. *ACM Transactions on Architecture and Code Optimization (TACO)*, 4(2), Jun 2007.

[33] J. M. Codina, F. J. Sanchez, and A. Gonzalez. A Unified Modulo Scheduling and Register Allocation Technique for Clustered Processors. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pp. 175-184., Spain, Sep. 2001.

[34] K. Kailas, A. Agrawala, K. Ebcioglu. CARS: A New Code Generation Framework for Clustered ILP Processors, In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, Jan 2001.

[35] E. Ozer, S. Banerjia, and T. Conte. Unified Assign and Schedule: A New Approach to Scheduling for Clustered Register File Microarchitectures. In *Proceedings of the 31st Annual IEEE/ACM International Symposium on Microarchitecture*, Dallas, Texas, USA, Nov. 1998.

[36] X. Guan and Y. Fei. Register File Partitioning and Recompilation for Register File Power Reduction, *ACM Transactions on Design Automation of Electronic Systems(TODAES)*, 15(3): 1-30, May 2010.

Je-Hyung Lee received the B.S. degree in School of Electronics Engineering from Kyungpook National University, Korea in February 1999, and the M.S. and Ph.D. degrees in School of EE&CS from Seoul National University, Korea in February 2001 and August 2009, respectively. He is currently a senior engineer in Samsung Electronics. He is working on compilers in mobile platforms and his research interests include compiler and VM performance optimizations in LLVM, GCC and JavaScript Engines.

Soo-Mook Moon received his Ph.D at the University of Maryland, College Park, in 1993. During 1992-1993, he worked at IBM Thomas J. Watson Research Center where he developed the IBM VLIW compiler. During 1993-1994, he was a software design engineer at the Hewlett-Packard Company in California Language Lab where he contributed to the development of an optimizing compiler for the PA-RISC CPUs. Since 1994, he has been with the faculty of the Seoul National University in the School of Electrical Engineering and Computer Science where he is now a full professor.

Jinpyo Park received his Ph.D at Seoul National University in 2003. He is now working at Samsung Electronics. His area of interest is SoC design, especially low power SoC design, bus architecture, and memory subsystem.

[37] [illegible] 2009.