Microarchitecture and Compiler Techniques for Dual Width ISA processors

by

Arvind Krishnaswamy

A Dissertation Submitted to the Faculty of the

Department of Computer Science

In Partial Fulfillment of the Requirements
For the Degree of

Doctor of Philosophy

In the Graduate College

The University of Arizona

2 0 0 6

Get the official approval page
from the Graduate College
before your final defense.

Statement by Author

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

Signed: _____

Table of Contents

List of Figures **6**

List of Tables **8**

Abstract **9**

Chapter 1. Introduction **10**
1.1. Code Size vs Performance Problem 11
1.2. Dual Width ISA ARM Processors 12
1.3. The Approach 14
1.4. Organization 15

Chapter 2. Dynamic Instruction Coalescing **16**
2.1. Dynamic Instruction Coalescing With Augmenting eXtensions 16
2.2. DIC Microarchitecture 18
2.3. Predicated Execution in AXThumb 26
2.4. AX Extensions to Thumb 28
2.4.1. ALU Instructions 29
2.4.2. Predication 32
2.4.3. MOV Instructions 33
2.4.4. Encoding of AX Instructions 37
2.5. Related Work 38
2.6. Summary 40

Chapter 3. Local Optimizations Using DIC **42**
3.1. Compiler Algorithms 42
3.1.1. Phase 1 - Predicated Code 43
3.1.2. Phase 2 - Peephole Optimizations 46
3.1.3. Phase 3 - Function Prologues and Epilogues 48
3.2. Profile Guided Approach for Mixed Code 51
3.2.1. BX/BLX instructions 52
3.2.2. Profile Guided Mixed Code Heuristic (PGMC) 53
3.3. Experiments 54
3.3.1. Performance of AXThumb 55
3.3.2. Comparison with Profile Guided Mixed Code 62
3.4. Summary 64

Table of Contents—*Continued*

Chapter 4. Global Optimization Using DIC **65**
4.1. Exposing Invisible Registers 66
4.2. Exploiting Exposed Registers 68
4.2.1. Initial Placement Points 70
4.2.2. Placement Ranges 72
4.2.3. Coalescing and Final Placement 75
4.3. Experiments 81
4.4. Discussion 87
4.5. Related Work 88
4.6. Summary 90

Chapter 5. Dynamic Eager Execution **91**
5.1. Delayed Branching and n-Wide Execution 91
5.1.1. Minimizing Branch Penalty 91
5.1.2. Maximizing Instruction Issue and Execution 94
5.2. Dynamic Eager Execution Microarchitecure 96
5.3. Experiments 101
5.4. Summary 104

Chapter 6. Conclusion **105**
6.1. Contributions 105
6.2. Future Work 107

References **109**

List of Figures

Figure 1.1. ARM vs Thumb Code 11

Figure 2.1. Thumb Implementation. 19
Figure 2.2. AXThumb Implementation. 20
Figure 2.3. State Transitions of the Instruction Buffer. 24
Figure 2.4. Delivering Instructions to Decode Ahead for Overlapped Execution. 24
Figure 2.5. Predication in AXThumb. 27

Figure 3.1. Predication 45
Figure 3.2. Phase 2 49
Figure 3.3. SetAllHigh AX transformation 51
Figure 3.4. Replacing Thumb Sequence by ARM Sequence. 53
Figure 3.5. Normalized Instruction Counts 57
Figure 3.6. Normalized Cycle Counts 58
Figure 3.7. Normalized Code Size 60
Figure 3.8. Normalized I-Cache Energy 61

Figure 4.1. Register Operand Access 67
Figure 4.2. Use of `SetMask`. 68
Figure 4.3. Normalized Code Size 69
Figure 4.4. Step 1: Initial Placement Points Determination. 70
Figure 4.5. Initial Placement Points. 71
Figure 4.6. Propagating Initial Placement Points Backwards to Build Placement Ranges. 72
Figure 4.7. Placement Ranges. 73
Figure 4.8. Final Placement Points. 75
Figure 4.9. Splitting Placement Ranges into Placement Paths. 77
Figure 4.10. Overlap Graph. 78
Figure 4.11. Clique Selection and Final Placement. 78
Figure 4.12. Clique Selection. 79
Figure 4.13. Coalescing and Final Placement. 80
Figure 4.14. Normalized Code Size 83
Figure 4.15. Normalized Instruction Counts. 85
Figure 4.16. Normalized Cycle Counts. 86

Figure 5.1. Delayed Branching Best Case 93
Figure 5.2. Delayed Branching Worst Case 93
Figure 5.3. In-order Superscalars vs VLIW Processors 95
Figure 5.4. Cases for Dynamic Delayed Branching 96

List of Figures—*Continued*

Figure 5.5. Cases for Dynamic 2-Wide Execution 96
Figure 5.6. Dynamic Eager Execution Microarchitecture 98
Figure 5.7. Extending the instruction buffer to 48 bits 100
Figure 5.8. Cycle counts for traditional and DEE Thumb 103

List of Tables

Table 2.1. Different Buffer States. 23
Table 2.2. Description of ARM/Thumb Instructions Used 29

Table 3.1. Benchmark Description 56
Table 3.2. Usage of Different AX Instructions. 63

Table 4.1. Percentage of Executed MOVs Eliminated. 84

Abstract

Embedded processors have to execute programs under the constraints of limited resources such as memory and power. As a result, code size becomes as important a metric as performance when evaluating applications written for the embedded domain. Existing techniques improve one program metric at the cost of the other. Simultaneously achieving good code size and performance is a challenging problem. This dissertation proposes compiler and microarchitectural techniques that address this problem.

Dual-Width ISA processors provide a platform with two instruction sets - a 32-bit instruction set yielding fast programs and a 16-bit instruction set yielding small programs. The techniques described here exploit properties of dual-width ISA processors to bridge the gap between the small programs and the fast programs by improving the performance of 16-bit programs, yielding small *and* fast programs. An integrated microarchitectural/compiler framework (Dynamic Instruction Coalescing) and a purely microarchitectural framework (Dynamic Eager Execution) are proposed. Dynamic Instruction Coalescing introduces a new kind of instruction - an Augmenting eXtension or AX. AX instructions are dynamically coalesced with the succeeding instruction at no cost. Efficient compiler techniques are proposed to use AX instructions to perform local and global optimizations that improve performance without negatively affecting code size. Dynamic Eager Execution is a microarchitecture that improves the performance of 16-bit programs by eagerly executing instructions. This framework comprises two techniques namely Dynamic Delayed Branching and Dynamic 2-wide Execution. The first improves branch behavior and the other seeks to improve program execution by simultaneously issuing multiple instructions.

Chapter 1
Introduction

Applications written for the embedded domain have to perform under the constraints of limited memory and limited energy. While these constraints have always existed, current trends such as mobile computing and ubiquitous computing bring more and more complex applications to the embedded domain, making performance or speed of execution an important factor as well. For instance, we are now able to run resource intensive gaming and multimedia applications on memory constrained hand-held devices. This poses the challenging task of simultaneously achieving small code size and high performance.

Dual-Width ISA processors move one step towards addressing this problem by providing two instruction sets - one for compact code and the other for high performance. 32-bit embedded cores from the ARM and MIPS family for instance support a 16-bit instruction set in addition to the traditional 32-bit instruction set. This gives the processor the ability to adapt itself to different constraints for different applications. While this provides the programmer and/or compiler the flexibility to choose either small *or* fast code, it fails to provide the ideal case: small *and* fast code. To this end, in this dissertation, microarchitectural and compiler techniques are proposed to improve the performance of 16-bit code without negatively affecting code size in dual-width ISA processors.

We begin by looking at the code size and performance characteristics of dual width ISA processors. Without delving into much detail the next section describes the key characteristics of dual width ISA processors. In particular, we will look at the shortcomings and tradeoffs of the the two instruction sets. This section is followed by a description and rationale of our overall approach to solving the problem of achieving

both small code size and good performance. This chapter ends with a description of the organization of the rest of this dissertation.

1.1 Code Size vs Performance Problem

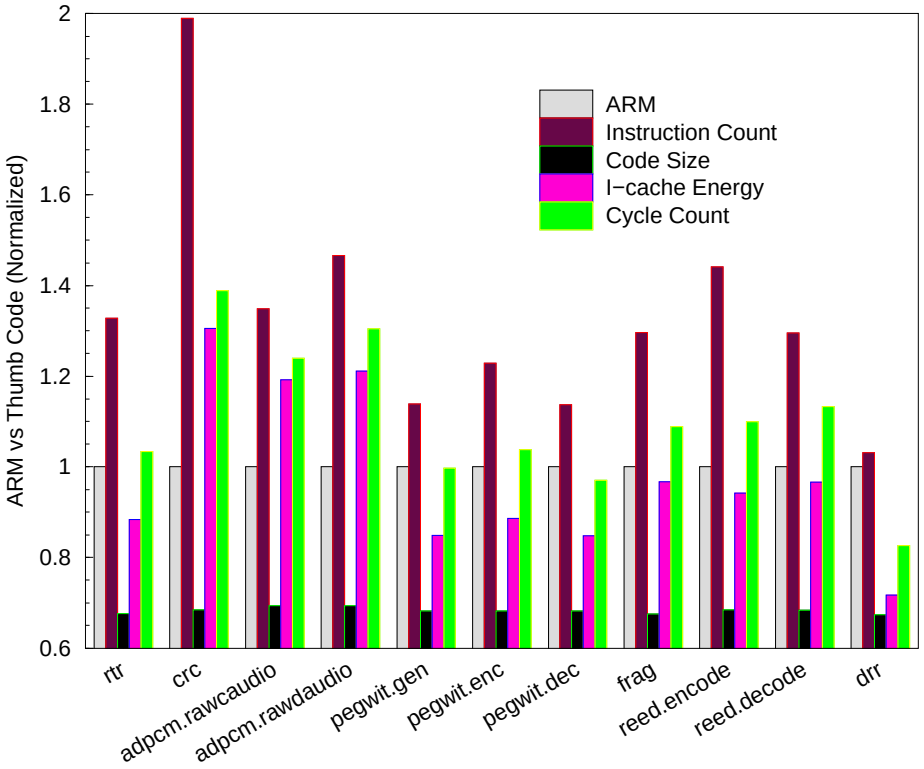


FIGURE 1.1. ARM vs Thumb Code

In this section we look at a quantitative comparison between 16-bit Thumb code and 32-bit ARM code. This comparison and the techniques described in this dissertation are in the context of the ARM architecture which is representative of dual width ISA architectures. The problems and techniques apply to other dual width ISA processors like the MIPS and SuperH family of processors as well.

The data in Figure 1.1 compares the ARM and Thumb codes along four metrics: instruction count, code size, I-cache energy, and cycle count. The data is normalized to ARM code, where the gray bars indicate the ARM code which have the value

of 1. All other values are the corresponding Thumb values divided by the ARM values to give the Normalized Thumb values. In addition to measuring code size and performance we also measure energy spent in the instruction cache since the I-cache contributes a significant portion of the total processors energy. As we can see, the number of instructions executed by Thumb code is significantly higher even though the Thumb code size is significantly smaller. The increase in instruction counts ranges from 3% to 98% while code size reduction ranges from 29.83% to 32.45% (Segars et al. [31] also report a 30% code size reduction). The substantial increase in the number of instructions executed by the Thumb code more than offsets the improved I-cache behavior of the Thumb code. Executing more instructions results in increased energy consumption. Moreover, while the cycle counts for Thumb are significantly higher compared to their ARM counterparts so are the I-cache energy values in some cases. Hence even from an energy standpoint, it is not only important to keep code size low, it also important to keep dynamic instruction and cycle count low. Future experiments will focus mainly on code size and performance. The reader is referred to [19] for a more in-depth quantitative comparison of ARM and Thumb code.

1.2 Dual Width ISA ARM Processors

More than 98% of all microprocessors are used in embedded products, the most popular among them being the ARM family of embedded processors [15]. The ARM processor core is used both as a macrocell in building application specific system chips and standard processor chips [9] (e.g., ARM810, StrongARM SA-110 [14], XScale [13]). This section provides a brief overview of the dual width instruction set support in the ARM architecture.

The ARM architecture is a 32-bit RISC architecture [9] supporting two instruction sets, a 32-bit ARM instruction set and a 16-bit Thumb instruction set. The processor has a fixed fetch bandwidth of 32-bits and is an in-order single issue processor. Cor-

responding to the two instruction sets there are two execution states. In ARM state 32-bit instructions are executed and in Thumb state 16-bit instructions are executed. The ARM ISA (instruction set architecture) supports a 3-address format, supports predicated execution and can access all 16 32-bit registers. In Thumb state however, instructions are restricted to a 2-address format, can access only eight 32-bit registers in most cases and do not support predicated execution. The Thumb instruction set has limited expressive power compared to the ARM instruction set. For example, in an ARM instruction it is possible to specify a shift operation along with an ALU operation in the same 32-bit instruction but in Thumb state two instructions are required.

Since the full expressiveness of the 32-bit ARM ISA is not always necessary, one can achieve considerable code size reductions using 16-bit Thumb instructions. The Thumb version of an application is on average 30% smaller than its 32-bit ARM counterpart [31]. It should be noted that using Thumb code, with every 32 bits fetched, the processor fetches 2 Thumb instructions. Hence the processor needs to fetch a word only every other cycle, reducing the amount of energy spent on fetching instructions from the instruction cache. Considering that a lot of energy is spent in the instruction cache (the cache is fully associative requiring multiple simultaneous lookups), this reduction is significant. Thumb code being small also provides a good locality of reference. Hence there are fewer cache misses in Thumb code compared to ARM code. While there is a significant reduction in code size and energy when we use Thumb code, we lose a considerable amount of performance. This is because for the same task we need many more Thumb instructions compared to the number of ARM instructions. This loss is incurred in spite of the good locality provided by Thumb code. By proposing integrated microarchitectural and compiler techniques we show how one can exploit dual width ISA processors to provide high performance small code.

1.3 The Approach

To achieve our goal of small *and* fast code, one can think of two approaches. We can start with fast 32-bit code and try to reduce its code size. Or we can start with 16-bit code and try to improve its performance. Lets see why it is better to choose the latter option.

Code size reduction can be handled in two ways. First, we could compress the code using existing or custom compression algorithms and provide architectural support for dynamic decompression. Several such approaches have been suggested in [37] [24] [23]. While code size reductions comparable to Thumb have been achieved using these techniques they have a performance overhead as well as significant additional hardware. Moreover, such architectures are not backward compatible. Similar compression techniques have been employed by compilers as suggested in [5]. Second, we could use compiler based code compaction techniques [6] that employ compiler transformations to reduce code size. While these techniques effectively reduce code size, they can be applied to both 32-bit and 16-bit code. In other words, they do not exploit any specific characteristics of the ISA.

Improving the performance of 16-bit code is the right approach for two reasons. First, the resources already present in dual width ISA processors are not fully utilized during 16-bit execution. We can exploit artifacts such as the 32-bit fetch bandwidth and higher order registers more effectively to improve performance. Second, we can study the tradeoffs between the two instructions sets and discover the shortcomings of the 16-bit instruction set. We can then address these ISA specific shortcomings through microarchitectural/ISA/compiler techniques.

In addition to addressing the shortcomings of the Thumb ISA techniques this dissertation introduces techniques that use existing resources to provide features in Thumb state not originally available in ARM state. The idea of Augmenting eXtensions is introduced. Augmenting eXtensions are instructions that carry some aug-

menting information and are fully processed early in the processor pipeline. A microarchitectural technique, Dynamic Instruction Coalescing, is proposed to effectively execute these AX instructions. The necessary ISA changes and compiler algorithms to effectively use AX instructions are also described. Additionally, a purely microarchitectural technique, dynamic eager execution, is proposed. *DEE* uses existing resources of dual width ISA processors to improve the performance of 16-bit code by executing instructions eagerly to provide a form of delayed branching and 2-wide execution.

1.4 Organization

The techniques used to improve the performance of 16-bit code is described in 4 chapters. Chapter 2 introduces the Dynamic Instruction Coalescing Microarchitecture and Augmenting eXtensions. The shortcomings of the Thumb ISA are studied in detail and the notion of augmenting instructions is introduced. Dynamic Coalescing enables these instructions to execute at zero cost. Support for Predication is also provided through this framework. Chapter 3 describes the compiler algorithms needed to perform local optimizations using the AX instructions introduced in Chapter 3. A comparison with the Profile Guided Mixed Code Approach is also made. Chapter 4 introduces a new AX instruction that changes the way the register file is accessed. This enables all registers from the register file to be allocated. The necessary microarchitecture changes needed to support this new instruction are described. The compiler algorithms required to perform global optimizations using this new AX instruction are also described. Chapter 5 introduces a purely architectural technique to improve the performance of 16-bit Thumb code. The Dynamic Eager Execution Framework is introduced here. We finally conclude in Chapter 6.

Chapter 2
Dynamic Instruction Coalescing

In this chapter, Dynamic Instruction Coalescing and Augmenting eXtension Framework is described. This framework serves as the platform for the techniques described in the next two chapters. The 16-bit instruction set, due to limited encoding space, can encode only a subset of the 32-bit instruction set. As a result in many cases two 16-bit instructions are required to achieve the same effect as one 32-bit instruction. This is one of the primary causes for the poorer performance of 16-bit code compared to 32-bit code. To support 32-bit execution dual width ISA processors need a 32-bit fetch bandwidth. This allows two 16-bit instructions to be fetched every cycle. Thus, during 16-bit execution the processor has the ability to lookahead one instruction. This property is exploited to execute two 16-bit instructions for the price of one. The concept of an Augmenting eXtension is introduced. We study code generated for ARM and Thumb to uncover shortcomings in the Thumb ISA. By tracing the performance bottlenecks in Thumb code back to the ISA, appropriate AX instructions are proposed to improve performance when used in the DIC microarchitecture.

2.1 Dynamic Instruction Coalescing With Augmenting eXtensions

The DIC and AX framework is the microarchitecture and ISA portion of the integrated compiler and microarchitecture proposal to improve the performance of Thumb code. A hardware based runtime instruction coalescing mechanism in the decode stage of the processor pipeline replaces pairs of 16-bit instructions by equivalent single ARM instructions for execution. The 16-bit pair consists of an Augmenting eXtension and a regular 16-bit Thumb instructions. Here we describe the coalescing hardware and

the Augmenting eXtensions.

To illustrate the key concepts of our approach let us look at a simple example. The code below shows an ARM instruction which shifts the value in `reg2` before subtracting it from `reg1`. Since the shift cannot be specified as part of another Thumb ALU instruction, two Thumb instructions are required to achieve the effect of one ARM instruction. We would like to coalesce the two 16-bit instructions into one 32-bit instruction. While coalescing is relatively easy to carry out, detecting a legal opportunity for coalescing by examining the two Thumb instructions is in general impossible to carry out at run-time with simple hardware. In our example, the Thumb code uses a temporary register `rtmp`. If instruction coalescing is performed, `rtmp` is no longer needed; therefore its contents will not be changed. Hence, at the time of coalescing, the hardware must also determine that the contents of register `rtmp` will not be used after the Thumb sequence. Clearly this is in general impossible to determine since the next read or write reference to register `rtmp` can be arbitrarily far away.

Original ARM
`sub reg1, reg2, lsl #2`
Thumb
`lsl rtmp, reg2, #2` `sub reg1, rtmp`
AXThumb
`setshift lsl #2` `sub reg1, reg2`

Since the coalescing opportunity cannot be detected in hardware we rely on the compiler to recognize such opportunities and communicate them to the hardware through the use of *Augmenting eXtensions* (AX). In the AXThumb code shown above, the first instruction is an augmenting instruction which is not executed; it is always coalesced in the decode stage with the instruction that immediately follows it, to generate a single ARM instruction for execution. In the above example, the augmenting instruction `setshift` merely carries the shift type and shift amount, which

is incorporated in the subsequent instruction to create the required ARM instruction for execution.

We make the design choice that each Thumb instruction can be *augmented* only by a single AX instruction. As a result we are guaranteed that an AX instruction is always preceded and followed by a Thumb instruction. While it is possible to support a more flexible mechanism which allows an instruction to be augmented by multiple AX instructions, this is not useful as it does not speed up the execution of the Thumb code. The reason for this claim will become clear when we discuss the microarchitecture design in greater detail.

It should be noted that the code size of all three instruction sequences is the same (i.e., 32 bits). However, only the AXThumb sequence satisfies the desired criteria as it results in the execution of a single equivalent ARM instruction and is made up of 16-bit instructions. Thus, the AXThumb code is 16-bit code that runs like the ARM code.

We have introduced the basic idea behind our approach. Next, we describe in detail the realization of this idea. First, we describe the modified microarchitecture that is capable of executing AXThumb code in a manner such that coalescing does not introduce additional pipeline delays. Second, we describe the complete set of AX instructions and the rationale behind the design of these instructions.

2.2 DIC Microarchitecture

Our work is based upon the StrongARM SA-110 pipeline which consists of five stages: (F) instruction fetch; (D) instruction decode and register read; branch target calculation and execution; (E) Shift and ALU operation, including data transfer and memory address calculation; (M) data cache access; and (W) result write-back to register file. It performs in-order execution and does not employ branch prediction. The Thumb instruction set is easily incorporated into an ARM processor with a few

simple changes. The basic instruction execution core of the pipeline remains the same as it is designed to execute only ARM instructions. A Thumb instruction decompressor, which translates each Thumb instruction to an equivalent ARM instruction, is added to the instruction decode stage. Since the decoder is simple and does little work, this addition does not increase the cycle time.

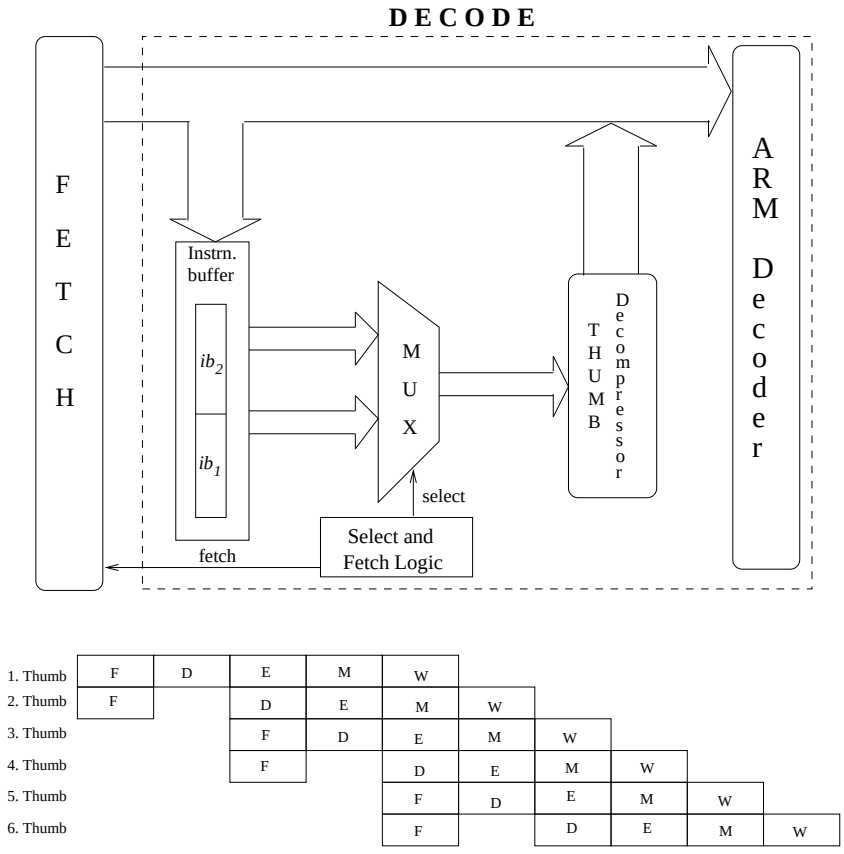


FIGURE 2.1. Thumb Implementation.

Before we describe our design of the decode stage, let us first review the original design of the decode stage, which allows the ARM processor to execute both ARM and Thumb instructions. As shown in Figure 2.1, the fetch capacity of the processor is designed to be 32 bits per cycle so that it can execute one ARM instruction per cycle. In the ARM state, a 32-bit instruction is directly fed to the ARM decoder.

However, in the Thumb state, the 32 bits are held in an *instruction buffer*. The two Thumb instructions in the buffer are selected in consecutive cycles and fed into the Thumb decompressor, which converts the Thumb instruction into an equivalent ARM instruction and feeds it to the ARM decoder. Every time a word is fetched we get two Thumb instructions. Hence, fetch needs to be carried out only in alternate cycles.

The key idea of our approach is to process an AX instruction simultaneously with the processing of the immediately preceding Thumb instruction. What makes this achievable is the extra fetch capacity already present in the processor.

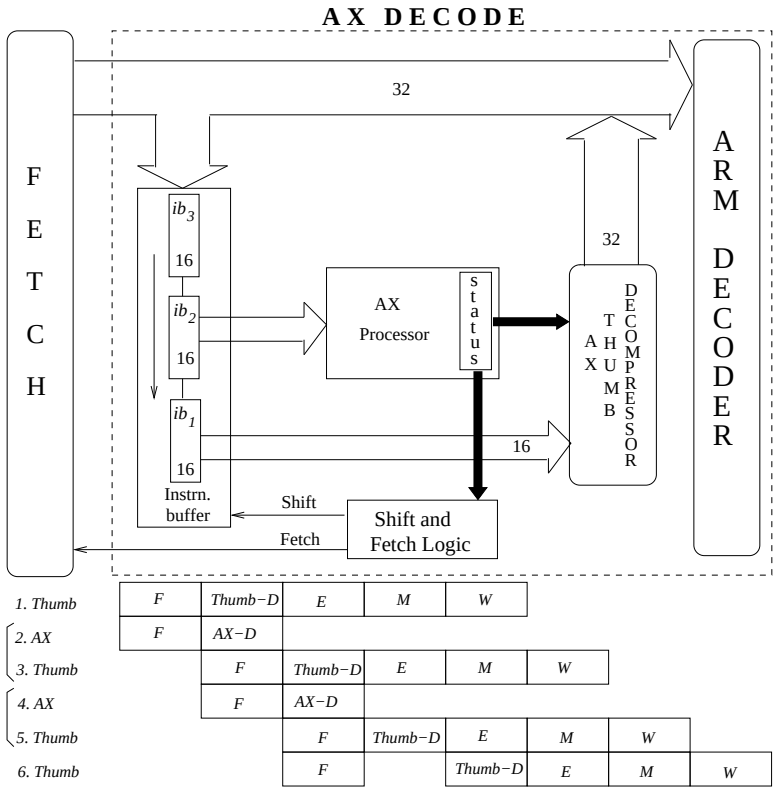


FIGURE 2.2. AXThumb Implementation.

The overall operation of the hardware design shown in Figure 2.2 is as follows. The *instruction buffer* in the decode stage is modified to exploit the extra fetch bandwidth and keep at least two instructions in the buffer at all times. Two consecutive instructions, one Thumb instruction and a following AX instruction, can be simultaneously

processed by the decode stage in each cycle. The AXThumb instruction is processed by the *AX processor* which updates the *status* field to hold the information carried by the AX instruction for augmenting the next instruction in the following cycle. The Thumb instruction is processed by the *AXThumb decompressor* and then the *ARM decoder*. The decompressor is enhanced to use both the current Thumb instruction and the status field contents modified by the immediately preceding AX instruction in the previous cycle, if any, to generate the *coalesced* ARM instruction. The status field is read at the beginning of the cycle for use in generation of the coalesced ARM instruction and overwritten at the end of the cycle if an AX instruction is processed in the current cycle. The status field can be implemented as a 28-bit register. Hence, during a context switch it is sufficient to save the state of this status register along with other state to ensure correct execution when this context resumes. The format of this status register is described along with the encodings of AX instructions in Section 2.2.4.

There are three important points to note about the above operation. First, as shown by the pipeline timing diagram in Figure 2.2, in the above operation *no extra cycles* are needed to handle the AX instructions. Each sequence (pair) of AX and Thumb instructions complete their execution one cycle after the completion of the preceding Thumb instruction. Second the above design ensures that there is *no increase in the processor cycle time*. The AX processor's handling of the AX instruction is entirely independent of handling of the Thumb instruction by the decode stage. In the pipeline diagram Thumb-D and AX-D denote handling of Thumb and AX instructions by the decode stage respectively. In addition, the path taken by the Thumb instruction is essentially the same as the original design: the Thumb instruction is first decompressed and then decoded by the *ARM decoder*. The only difference is the modification made to the decompressor to make use of the *status* field information and carry out *instruction coalescing*. However, this modification does not significantly increase the complexity of the decompressor as the generation of an ARM instruction

through coalescing of AX and Thumb instructions is straightforward. An AX instruction essentially predetermines some of the bits of the ARM instruction generated from the following Thumb instruction. This should be obvious for the `setshift` example already shown. The other AX instructions that are described in detail in the next section are equally simple. Finally it should now be clear why we do not allow two AX instructions to augment a Thumb instruction. Only a single AX instruction can be executed for free. If two consecutive AX instructions are allowed, their execution will add a cycle to the program's execution. Moreover, one AX instruction is sufficient to augment one Thumb instruction as it can carry all the required information. Hence, even in the case where we have more bandwidth (e.g., 64 bits), using more than one AX instruction to augment a Thumb instruction is not useful.

The instruction buffer and the filling of this buffer by the instruction fetch mechanism are designed such that, in the absence of taken branches, the instruction buffer always contains at least two instructions. The buffer can hold up to three consecutive instructions. Thus, it is expanded in size from 32 bits (ib_1 and ib_2) in the original design to 48 bits (ib_1, ib_2, and ib_3). As shown later, this increase in size is needed to ensure that at least two instructions are present in the instruction buffer. Of the three consecutive program instructions held in ib_1, ib_2 and ib_3, the first instruction is in ib_1, second is in ib_2 and third one is in ib_3. The instruction in ib_1 is always a Thumb instruction which is processed by the Thumb decompressor and the ARM decoder. The instruction in ib_2 can be an AX or a Thumb instruction and it is processed by the AX processor. If this instruction is an AX instruction then it is completely processed, and at the end of the cycle, instructions in both ib_1 and ib_2 are consumed; otherwise only the instruction in ib_1 is consumed. The remaining instructions in the buffer, if any, are *shifted* by 1 or 2 entries so that the first unprocessed instruction is now in ib_1. The fetch deposits the next two instructions from the instruction fetch queue into the buffer at the beginning of the next cycle if at least two entries in the buffer are empty. Therefore, essentially there are two cases: either the two instructions are

deposited in (ib_1, ib_2) or in (ib_2, ib_3).

TABLE 2.1. Different Buffer States.

State	ib1	ib2	ib3
S1	-	-	-
S2	T	-	-
S3	T	T	-
S4	T	A	-
S5	T	T	T
S6	T	A	T

We summarize the above operation of the instruction buffer using a state machine. Table 2.1 describes the various states of the buffer depending upon its contents – a `T` indicates a Thumb instruction and an `A` indicates an AX instruction. The states are defined such that they distinguish between the number of instructions in the buffer – $S1$, $S2$, $S3/S4$, and $S5/S6$ correspond to the presence of 0, 1, 2, and 3 instructions in the buffer respectively. Pairs of states $(S3, S4)$ and $(S5, S6)$ are needed to distinguish between the absence and presence of an AX instruction in ib_2. This is needed because the presence of an AX instruction results in coalescing while its absence means that no coalescing will occur. Given these states, it is easy to see how the changes in the buffer state occur as instructions are consumed and a new instruction word is fetched into the buffer whenever there is enough space in it to accommodate a new word. The state diagram is summarized in Figure 2.3.

Now we illustrate the need to expand the instruction buffer to hold up to three instructions. In Figure 2.4(a) we show a sequence in which the AX instruction(s) cannot be processed in parallel with the preceding Thumb instruction(s) as only after the preceding Thumb instruction(s) are processed can the instruction fetch deposit an additional pair of instructions into the buffer. Therefore, the advantage of providing AX instructions is lost. On the other hand, in Figure 2.4(b), when we expand the buffer to 48 bits, the instructions are deposited by the fetch sooner, thereby causing

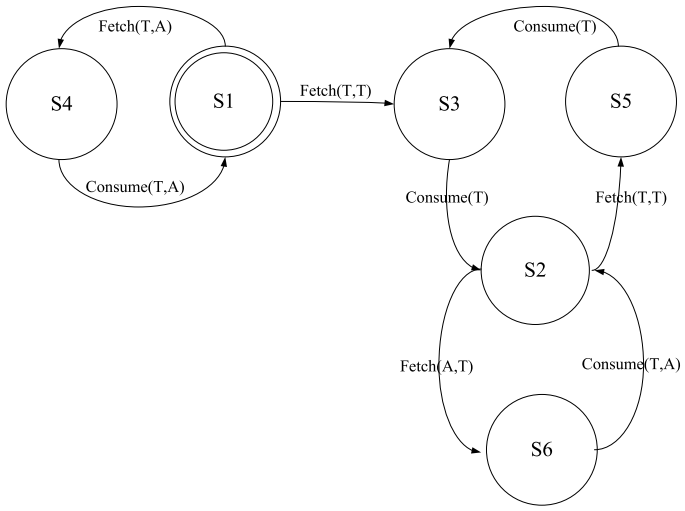
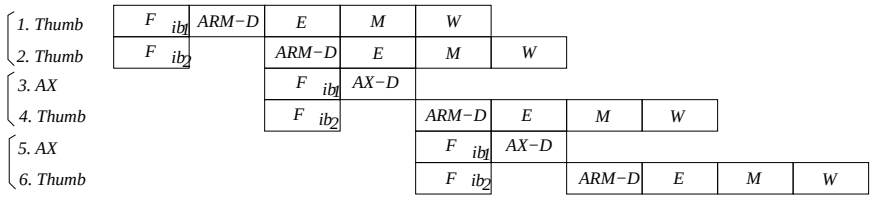
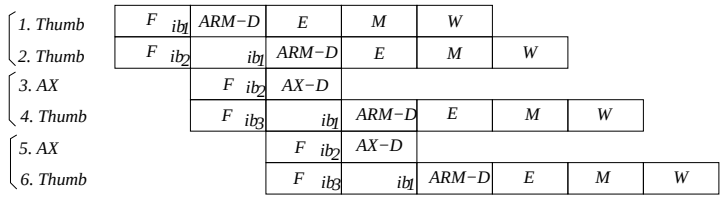


FIGURE 2.3. State Transitions of the Instruction Buffer.

the AX instruction(s) and the preceding Thumb instruction(s) to be simultaneously present in the buffer. Hence, the AX instructions are now handled for free.



(a) 32 bit Instruction Buffer.



(b) 48 bit Instruction Buffer.

FIGURE 2.4. Delivering Instructions to Decode Ahead for Overlapped Execution.

Next, we show how it is ensured that whenever an instruction is found in ib_1, it is always a Thumb instruction. If the instruction was shifted from ib_2 it must be a Thumb instruction as the AX processor has concluded that it is not an AX instruction. If the instruction was shifted from ib_3, it must be a Thumb instruction. This is because

in the preceding cycle the instruction in ib_2 must have been successfully processed, meaning that it was an AX instruction which implies the next instruction, (i.e., the one in ib_3), must be a Thumb instruction. The final case is when the fetch directly deposits the next two instructions into (ib_1, ib_2). Clearly the instruction in ib_1 is not examined by the AX processor in this case. Therefore, it must be guaranteed that whenever the instruction buffer is empty at the end of the decode cycle, the next instruction that is fetched is a Thumb instruction.

In the absence of branches the above condition is satisfied. This is because at the beginning of the decode cycle the buffer definitely contains two instructions. For it to be empty the two instructions must be simultaneously processed. This can only happen if the instruction in ib_2 was an AX instruction which implies that the next instruction is a Thumb instruction.

In the presence of branches, following a taken branch, the first fetched instruction is also directly deposited into ib_1. We assume that the instruction at a branch target is a Thumb instruction; hence, it can be directly deposited into ib_1 as examination of the instruction by the AX processor is of no use. The compiler is responsible for generating code that always satisfies this condition. The reason for making this assumption is that there is no advantage of introducing an AX instruction at a branch target. Only an AX instruction that is preceded by another Thumb instruction can be executed for free. If the instruction at a branch target is an AX instruction, and control arrives at the target through a taken branch, then the processing of the AX instruction by the AX processor can no longer be overlapped with the immediately preceding instruction that is executed, that is, the branch instruction. This is because the AX instruction can only be fetched after the outcome of the branch is known.[1] Therefore, the execution of the AX instruction actually adds a cycle to the execution. In other words, the benefit of introducing the AX instruction is lost. When an AXThumb

[1]Note that the ARM processor does not support delayed branching and therefore an AX instruction cannot be moved up and placed in the branch delay slot.

pair replaces a Thumb pair, the second Thumb instruction in the AXThumb pair need not be the same as the second Thumb instruction in the Thumb instruction pair. Hence, one cannot allow an AX instruction in ib_1 by issuing a nop when an AX instruction is found in ib_1. We rely on the compiler to schedule code in a manner that avoids placement of an AX instruction at a branch target. If this cannot be achieved through instruction reordering, the compiler uses a sequence of two Thumb instructions instead of using a sequence of an AX and Thumb instructions at the branch target.

2.3 Predicated Execution in AXThumb

While the original Thumb instruction set does not support predicated execution, we have developed a very effective approach to carry out predicated execution using AXThumb code which requires only a minor modification to the decode stage design just presented. Like instruction coalescing, this method also takes advantage of the extra fetch bandwidth already present in the processor. We rely on the compiler to place the instructions from the true and false branches in an *interleaved* manner as shown in Figure 2.5. Since the execution of a pair of instructions is mutually exclusive, i.e. only one of them will be executed, in the decode stage we select the appropriate instruction and pass it on to the decompressor while the other instruction is discarded.

A special AX instruction precedes the sequence of interleaved instructions. This instruction communicates the predicate in form of a *condition flag* which is used to perform instruction selection from an interleaved instruction pair. If the condition flag is set, the first instruction belonging to each interleaved pair is executed; otherwise the second instruction from the interleaved pair is executed. Therefore, the compiler must always interleave the instructions from the true path in the first position and instructions from the false path in the second position. The special AX

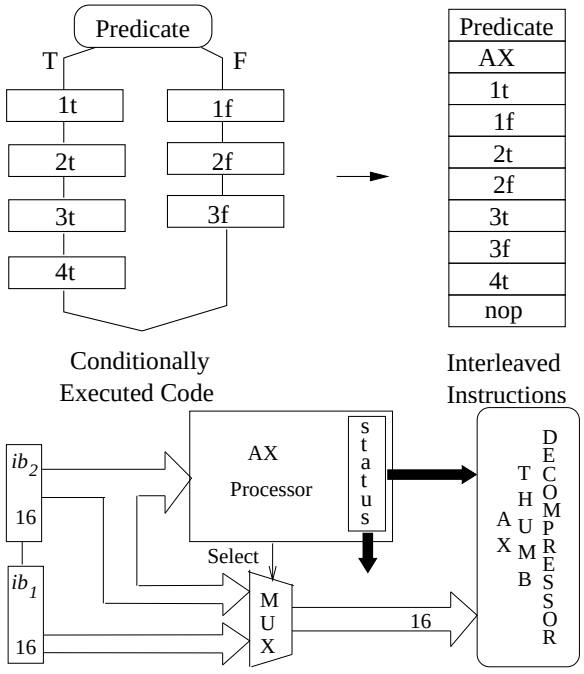


FIGURE 2.5. Predication in AXThumb.

instruction also specifies the count of interleaved instructions pairs that follow it. The AX processor uses this count to continue to stay in the predication mode as long as necessary and then switches back to the normal selection mode. The selection of an instruction from each instruction pair is carried out by using a minor modification to the original design as shown in Figure 2.5. Instead of directly feeding the instruction in ib_1 to the decompressor, the multiplexer selects either the instruction from ib_1 or ib_2 depending upon the predicate as shown in Figure 2.5. The select signal is generated by the AX processor. For correct operation, when not in predication mode, the select signal always selects the instruction in ib_1.

For this approach to work, each interleaved instruction pair should be completely present in the instruction buffer so that the appropriate instruction can be selected. This condition is guaranteed to be always true as the interleaved sequence is preceded by an AX instruction. Following the execution of the AX instruction there will be at least two empty positions in the instruction buffer which will be immediately filled by

the fetch. It should be noted that the `setpred` instruction essentially performs the function of setting bits in a predicate register which is part of the status register. The `setpred` instruction is slightly different from other AX instructions in that it does not enable any sort of instruction coalescing. As a result, it does not require the extra buffer length. Hence, this style of predication could be implemented independent of the rest of AX processing, by suitably modifying the fetching of instructions.

The above approach for executing predicated code is more effective than doing so in the ARM state. In ARM state the 32-bit instructions from the true and false paths are examined one by one. Depending on the outcome of the predicate test, instructions from one of the branches are executed while the instructions from the other branch are essentially converted into *nop*s. Therefore, the number of cycles needed to execute the instructions is at least equal to the sum of the instructions on the true and false paths. In contrast the number of cycles taken to execute the AXThumb code is equal to the number of interleaved instruction pairs. Note that this advantage is only achievable because in Thumb state instructions arrive in the decode stage early while the same is not true for ARM.

2.4 AX Extensions to Thumb

The AX extension to Thumb consists of eight new instructions. These instructions were chosen by studying ARM and Thumb codes of benchmarks and identifying commonly occurring sequences of Thumb instructions which were found to correspond to shorter ARM sequences of instructions. We describe these instructions and illustrate their use through examples of typical situations that were encountered. We categorize the AX instructions according to the types of instructions whose counts they affect the most. The following discussion will also make clear the differences in the ARM and Thumb instruction sets that lead to poorer quality Thumb code. We then show how we use exactly one free instruction in the free opcode space of the

Thumb instruction set to implement AX instructions. We also give the format of the 28-bit status register that is used during AX processing. A brief description of the ARM/Thumb instructions used here is shown in Table 2.2.

Table 2.2. Description of ARM/Thumb Instructions Used

Name	Description
`str`	Store to memory
`ldr`	Load from memory
`push`	Push contents onto stack
`pop`	Pop contents from stack
`b`	Unconditional Branch
`b[cond]`	Conditional Branch eg. `beq`
`and`	Logical AND
`neg`	Negates value and stores in destination
`mov`	Move contents between registers
`add`	Arithmetic Add
`sub`	Arithmetic Subtract
`lsl`	Logical Shift Left

2.4.1 ALU Instructions

There are specific differences in the ARM and Thumb instruction sets that cause additional ALU instructions to be generated in the Thumb code. There are three critical differences we have located and to compensate for each of three weaknesses in the Thumb instruction set we have designed a new AX instruction. ARM instructions are able to specify negative immediates, shift operations that can be folded into other ARM instructions, and certain kinds of compares that can be folded with other ARM instructions. None of these three features are available in the Thumb instruction set. The new AX instructions are as follows.

Negative Immediate
`setimm #constant`
Folded Shift
`setshift shifttype shiftamount`
Folded Compare
`setsbit`

Negative Immediate Offsets. The example shown below, which is taken from versions of the ARM and Thumb codes of a function in `adpcm_coder`, illustrates this problem. The constant negative offset specified as part of the `str` store instruction in ARM code is placed into register `rtmp` using the `mov` and `neg` instructions in the Thumb mode. The address computation of `rbase + rtmp` is also carried out by a separate instruction in the Thumb state. Therefore, one ARM instruction is replaced by four Thumb instructions.

Original ARM
`str rsrc, [rbase, -#offset]`
Corresponding Thumb

`mov rtmp, #offset`
`neg rtmp`
`add rtmp, rbase`
`str rsrc, [rtmp, #0]`

AXThumb
`setimm -#offset`
`str rsrc, [rbase, _]`
Coalesced ARM
`str rsrc, [rbase, -#offset]`

The AX instruction `setimm` is used to specify the negative operand of the instruction that immediately follows it. For our example, the `setimm` is generated immediately preceding the `str` instruction. When an `str` instruction immediately follows a `setimm` instruction, the constant offset is taken from the `setimm` and whatever constant offset that may be directly specified in the `str` instruction is ignored. In the decode stage the `setimm` and `str` are coalesced to generate the equivalent ARM instruction as shown above.

Shift Instructions. The setshift instruction has been shown through our example at the beginning of section 2. We describe one more use here. A shift operation folded with a MOV instruction is often used in ARM code to generate *large immediate constants*. An immediate operand of a MOV instruction is a 12 bit entity which is

divided into an 8-bit *immediate* constant and a 4-bit *rotate* constant. The eight bit entity is expanded to 32 bits with leading zeroes and rotated by the *rotate* amount to generate a 32-bit constant. The *rotate* amount is multiplied by two before rotating right. In Thumb state the immediate operand is only 8 bits and therefore the *rotate* amount cannot be specified. An additional ALU instruction, an *lsl* is used to generate the large constant by shifting left using the rotate amount as shown below. In the AXThumb code `setshift` is used to eliminate the extra shift instruction through coalescing.

Original ARM
`mov reg1, #imm8.rotate4`

Corresponding Thumb
`mov reg1, #imm8` `lsl reg1, #rotate4'`, where rotate4' = 32 - 2 * rotate4.

AXThumb
`setshift #rotate4` `mov reg1, #imm8`

Coalesced ARM
`mov reg1, #imm8.rotate4`

Compare Instructions. In the ARM instruction set, MOV and ALU instructions contain an `s`-bit. If the `s`-bit is set, following the MOV or ALU operation, the destination register contents are compared with the constant value zero and certain flags are set which can later be tested. Thus, in ARM certain types of compares can be folded into other MOV and ALU instructions. As illustrated below, since Thumb does not support the `s`-bit, it must perform the comparison in a separate instruction. To overcome the above drawback we introduce the `setsbit` instruction which indicates that the `s`-bit of the instruction that immediately follows should be set when translation of Thumb into ARM takes place.

Original ARM
`movs reg1, reg2`

Corresponding Thumb
`mov reg1, reg2` `cmp reg1, #0`

AXThumb
`setsbit` `mov reg1, reg2`

Coalesced ARM
`movs reg1, reg2`

2.4.2 Predication

Lack of predication in Thumb is the reason for more branches in Thumb code compared to ARM code, as illustrated by the example below. The ARM code performs the compare; if `r3` contains zero then the two `subne` instructions turn into *nop*s while the other two `addeq` instructions are executed. The reverse happens if `r3` does not contain zero. In the corresponding Thumb code explicit branches are introduced to achieve conditional execution of instructions.

Original ARM
`cmp r3, #0`
`addeq r6, r6, r1`
`addeq r5, r5, r2`
`subne r6, r6, r1`
`subne r5, r5, r2`

Corresponding Thumb
`cmp r3, #0`
`beq .L13`
`sub r6, r1`
`sub r5, r2`
`b .L14`
`.L13: add r6, r1`
`add r5, r2`
`.L14: ...`

AXThumb
`cmp r3, #0`
`setpred eq, #2`
`add r6, r1`
`sub r6, r1`
`add r5, r2`
`sub r5, r2`

Coalesced ARM
`cmp r3, #0`
`subne r6, r6, r1`
`subne r5, r5, r2`
OR
`cmp r3, #0`
`addeq r6, r6, r1`
`addeq r5, r5, r2`

The new `setpred` instruction we introduce enables conditional execution of Thumb instructions. This instruction specifies two things. First it specifies the *condition* involved in predication (e.g., `eq`, `ne` etc.). Second it specifies the *count* of predicated instruction pairs that follow. Following the `setpred` instruction are pairs of Thumb instructions – the number of such pairs is equal to *count*. If the *condition* is true, the first instruction in each pair is executed; otherwise the second instruction each pair is executed.

```
setpred condition, #count
```

In our example, when we examine the AXThumb code, we observe that the condition in this case is `eq` and `count` is two since there are two pairs of instructions that are conditionally executed. If `eq` is true the first instruction in each pair (i.e., the `add` instruction) is executed; otherwise the second instruction in each pair (i.e., the `sub` instruction) is executed. Therefore, after the AXThumb instructions are processed by the decode stage the corresponding ARM instruction sequence generated consists of three instructions. The sequence contains either the `add` instructions or the `sub` instructions depending upon the `eq` flag. Clearly the sequence of instructions generated using our method is shorter than the original ARM sequence since it does generate *nop*s for the two instructions that are not executed. Note that this form of predication is restricted to small length branch hammocks due to the lack of encoding space to specify the length of the predicated block in the `setpred` instruction. Larger blocks can be predicated by using multiple `setpred` instructions.

This form of predication could also reduce the number of fetches from the I-cache. In the case shown below Thumb requires one more fetch than AXThumb code for every iteration of the outer loop L0. Also note that use of predication reduces the size by one instruction.

Thumb Code
L0: I0
beq L1
I1
b L2
L1: I2
L2: beq L0

AXThumb
L0: I0
setpred EQ 1
I1
I2
beq L0

2.4.3 MOV Instructions

We have identified three distinct reasons due to which extra move instructions are required in Thumb code. First most ALU Thumb instructions cannot directly reference values held in higher order (`r8 - r11`) registers. Second while ARM supports three address instruction format, Thumb uses a two address format and therefore requires

additional move instructions. Finally in Thumb ADD/MOV instructions the result register can be a higher order register but in this case an immediate operand is not allowed. Therefore, the immediate operand must be moved into a register before it can be used by the high register based Thumb ADD/MOV instruction. The following AX instructions are used to overcome the above drawbacks.

High Register Operand

`setsource Hreg`
`setdest Hreg`
`setallhigh`

Third Operand
`setthird reg`
Immediate Operand
`setimm #constant`

High Register Operands. Consider the example of a load below in which the base address is in a higher order register. While the ARM load instruction can directly reference this register, the Thumb code requires the base address to be moved to lower order register which can be directly referenced by a Thumb load instruction.

Original ARM
`ldr reg, [Hreg, #offset]`
Corresponding Thumb
`mov Lreg, Hreg` `ldr reg, [Lreg, #offset]`

AXThumb
`setsource Hreg` `ldr reg, [_, #offset]`
Coalesced ARM
`ldr reg, [Hreg, #offset]`

The instruction `setsource Hreg` is used to handle the above situation. The Thumb instruction that follows the `setsource Hreg` instruction makes use of `Hreg` as its source operand. After coalescing, the resulting ARM instruction is identical to the ARM instruction used in the ARM code. The `setdest Hreg` is used in a similar way.

The `push` instruction is used to carry out saving of registers at function boundaries. The ARM `push` instruction provides a 16-bit mask which indicates which registers should be saved and which are not to be saved. The corresponding Thumb `push`

instruction provides a 8-bit mask which corresponds to lower order registers. As a consequence, saving of higher order registers requires additional move instructions in Thumb code as illustrated by the example given below. While ARM code can use a single `push` instruction to save both lower order registers (`r4 - r7`) and higher order registers (`r8 - r11`), The Thumb code uses one `push` to save lower order registers, then moves contents of higher order registers into lower order registers, and then uses another push to save their contents.

Original ARM
`push {r4,.., r11}`

Corresponding Thumb

`push {r4, r5, r6, r7}`
`mov r7, r11`
`mov r6, r10`
`mov r5, r9`
`mov r4, r8`
`push {r4, r5, r6, r7}`

AXThumb

`push {r4, r5, r6, r7}`
`setallhigh`
`push {r0, r1, r2, r3}`

Coalesced ARM
`push {r4, r5, r6, r7}` `push {r8, r9, r10, r11}`

To address this problem we provide the `setallhigh` AX instruction. When this instruction precedes a Thumb `push` instruction, the 8-bit mask is interpreted to correspond to higher order registers. In absence of preceding `setallhigh` instruction the 8 bit mask in the Thumb `push` instructions corresponds to the lower order registers. The bit positions of registers `r0` through `r7` in the mask correspond to that of `r8` through `r15` respectively. The AXThumb code for the above example contains two `push` instructions, the first one saves the contents of lower order registers and the second one preceded by `setallhigh` saves the contents of higher order registers. The move instructions present in the Thumb code have been eliminated. The difference between original ARM code and coalesced ARM code is that original ARM requires only a single `push` instruction while the coalesced ARM code contains two `push` instructions. `setallhigh` can similarly be used for restoring registers in combination with `pop`. Note that the AXThumb code has fewer 16-bit instructions, reducing both the code size and I-cache fetches compared to Thumb code.

Third Operand. Additional move instructions are required to compensate for the lack of three address instruction format in Thumb. We introduce the `setthird reg` AX instruction to avoid the extra move instruction. When a Thumb instruction is a preceded by a `setthird reg` instruction, then `reg` is treated as the third address for the Thumb instruction as shown below. Following coalescing the impact of extra move instruction is entirely eliminated.

Original ARM
`add reg1, reg2, reg3`
Corresponding Thumb
`mov reg1, reg2` `add reg1, reg3`

AXThumb
`setthird reg3` `add reg1, reg2`
Coalesced ARM
`add reg1, reg2, reg3`

Immediate Operand. The Thumb ADD/MOV instructions can directly reference higher order registers. However, in these cases if the operand cannot be an immediate constant, requiring an an extra move as shown below.

Original ARM
`add Hreg1, Hreg1, #imm`
Corresponding Thumb
`mov rtmp, #imm` `add Hreg1, rtmp`

AXThumb

`setimm #imm`
`add Hreg1, _`
OR
`setdest Hreg1`
`add _, #imm`

Coalesced ARM
`add Hreg1, Hreg1, #imm`

We can use the `setimm` instruction already introduced earlier to avoid the move instruction as shown above. The immediate operand is incorporated into the Thumb instruction that follows the `setimm` instruction by the coalescing actions of the decode stage resulting in a single ARM instruction. Alternatively the `setdest` instruction can be used as shown above. In either case the coalesced ARM instruction is the same.

Original ARM
`and reg1, reg1, #imm`
Corresponding Thumb
`mov rtmp, #imm` `and reg1, rtmp`

AXThumb
`setimm #imm` `and reg1, _`
Coalesced ARM
`and reg1, reg1, #imm`

Another situation where extra move instructions are generated due to the presence of immediate operands is when bitwise boolean operations are used. Instructions for these operations cannot have immediate operands generating an extra move.

2.4.4 Encoding of AX Instructions

Not surprisingly there are very few unused opcodes available in Thumb. We have chosen one of these available opcodes to incorporate the AX instructions. Bits 10..15 are taken up by this unused opcode 101110 which now refers to AX. The remaining bits 0..9 are available for encoding the various AX instructions. Since there are eight AX instructions, three bits are needed to differentiate between them - we use bits 7..9 for this purpose. The operands are encoded in the remaining bits 0..6.

Unimplemented Thumb Instruction

101110	xxxxxxxxxx
[10..15]	[0..9]

AX Instructions

101110	AX opcode	AX operands
[10..15]	[7..9]	[0..6]

The details of how operands are encoded for the various instructions are given next. Depending upon the number of bits available, the constant fields in various instructions are limited in size. The immediate constant in `setimm` is 7 bits, shift amount in `setshift` 4 bits, and count in `setpred` is 3 bits. Finally, registers are encoded using 4 bits so we can refer to both higher and lower order registers in AX instructions.

Encodings

101110	setimm	#constant
[10..15]	[7..9]	[0..6]

101110	setshift	shifttype	shiftamount
[10..15]	[7..9]	[4..6]	[0..3]

101110	setsbit	-
[10..15]	[7..9]	[0..6]

101110	setpred	condition	count
[10..15]	[7..9]	[3..6]	[0..2]

101110	setsource	Hreg	-
[10..15]	[7..9]	[3..6]	[0..2]

101110	setdest	Hreg	-
[10..15]	[7..9]	[3..6]	[0..2]

101110	setallhigh	-
[10..15]	[7..9]	[0..6]

101110	setthird	reg	-
[10..15]	[7..9]	[3..6]	[0..2]

The format of the status register used in AX processing is shown below. The state set by the various AX instructions is saved in this register in the appropriate field depending on the AX instruction. During a context switch, the whole register is saved and upon restoration, AX processing can continue as before.

Status Register Format

enable AX	setpred	ctr	register operand	imm	shamt	shtype	S bit	setallhigh
[27]	[24..26]	[20..23]	[16..19]	[9..15]	[5..8]	[2..4]	[1]	[0]

2.5 Related Work

Most closely related work can be classified broadly into two areas: Code compression and Coalescing techniques. Previous work in the area of code compression consists of techniques to compact code, keeping performance loss to a minimum. The technique we describe in this paper, improves the performance of already compact code. Coalescing techniques have been employed at various stages: compile time, binary translation time and dynamically using hardware at run-time. All of the techniques were applied in the context of wide issue superscalar processors, using a considerable

amount of hardware resources. Our technique, uses a limited amount of hardware resources, making it viable for an embedded processor. Let us look at specific schemes, in the above mentioned areas.

Wolfe and Chanin [37] proposed a compressed code RISC processor, where cache lines are huffman encoded and decompressed on a cache miss. The core processor is oblivious to the compressed code, executing instructions as usual. Compression ratios of 70% were reported. Lekatsas and Wolf [24] used the above model and proposed new schemes for compression by splitting the instruction space into streams to achieve better compression ratios. A dictionary based compression scheme was proposed by Lefurgy et al. [23]. The technique assigns shorter encodings for common sequences of instructions. These encodings and the corresponding sequences are stored in a dictionary. At runtime, the decoder uses the dictionary to expand instructions and execute them. Debray and Evans [5] describe a purely software approach to achieving compact code. Profiles are used to find the frequently executed portions of the program. The infrequently executed parts are then compressed, making decompression overhead low while achieving good compression ratios.

We now turn to previous approaches to Instruction Coalescing. Qasem et al. [27] describe a compile time technique to coalesce loads and stores. They use a special swap instruction that swaps the contents of memory and registers. As a result they execute fewer instructions and also reduce memory accesses. The picojava processor [25] implements instruction folding to optimize certain operations on the stack. A stack cache holds the top 64 values of the stack enabling random access to any of the 64 locations. For instructions that can be folded, like arithmetic operations with operands in the stack cache, the processor performs instruction folding by generating a RISC like instruction. This avoids unnecessary stack operations. Hu and Smith [12] recently proposed instruction fusing for the x86, where they fuse micro-instructions generated by x86 instructions. The dynamic translator fuses two dependent instructions if possible, reducing the number of slots occupied in the scheduling window

and improving ILP as a result. Instruction Coalescing/Preprocessing has been used for trace caches where the stored traces are optimized at runtime by the hardware. Friendly et al. [8] described an optimization that combined dependent shift and add instructions. Jacobsen and Smith [16] describe instruction collapsing where a small chain of dependent instructions is collapsed into one compound instruction. Both of the above techniques optimize the traces stored in the trace cache. Dynamic instruction stream editing (DISE)[4] is a processor extension for customizing applications to the contexts in which they run by dynamically transforming the fetched instruction stream, feeding the execution engine an instruction stream with modified or added functionality. DISE has been used to provide runtime decompression to support compressed code. While DISE modifies the instruction stream, unlike DIC which coalesces uncompressed instructions for performance, DISE expands compressed code.

Finally researchers have recognized the advantages of augmenting instruction sets. Given an instruction set and an application, it is often the case that one can identify additional instructions that would help improve the performance of the application. Razdan and Smith [29] proposed an approach for enabling introduction of such instructions by providing programmable functional units. In contrast, our approach to augmenting Thumb instruction set is not application specific or adaptable. It is rather specifically aimed at reintroducing instructions that had been eliminated from the ARM instruction set in order to create the Thumb instruction set.

2.6 Summary

In this chapter we have described the microarchitectural component required to perform dynamic instruction coalescing of AX eXtensions. With minimal hardware modifications to the existing pipeline we showed how AX instructions can be coalesced at runtime with the following Thumb instruction. We studied the Thumb ISA to uncover various opportunities to replace Thumb pairs with AX-Thumb pairs. We

described several AX instructions that enable local optimizations of Thumb code by exploiting such opportunities. We described how one can encode all of the AX instructions using a single free 16-bit opcode. We also showed how one can implement predication in 16-bit code using the *setpred* AX instructions. This chapter has laid out the foundation for the next two chapters which describe compiler techniques to generate high performance 16-bit code.

Chapter 3
Local Optimizations Using DIC

Extensions to the 16-bit ISA called Augmenting eXtensions (AX) were introduced in the previous chapter. This chapter described compiler algorithms that use AX instructions for local optimizations. They serve to carry the extra information that could not be specified in one 16-bit instruction and originally needed another 16-bit instruction. These instructions use the lookahead capability to coalesce two 16-bit instructions into one 32-bit equivalent at runtime. AX instructions are processed entirely in the decode stage by coalescing them with the previous 16-bit instruction. Hence they serve as a zero cycle 16-bit instruction that speeds up execution. The compiler is responsible for discovering opportunities for such local optimizations and inserting the appropriate AX instructions. We will look at the various local optimizations the compiler performs using AX instructions.

3.1 Compiler Algorithms

AXThumb transformations are performed as a postpass, after the compiler has generated object code. The transformation which involves detecting and replacing sequences of Thumb code with corresponding AXThumb code consists of three phases. Each of the three phases deals with a particular kind of AXThumb transformation. The first phase handles predication of Thumb code using the `setpred` AX instruction. The second phase handles the generic case for AX transformations like the example used to describe instruction coalescing. The third phase handles the `setallhigh` AX instruction used to eliminate unnecessary moves at function prologues and epilogues. While we present a postpass approach to generate AXThumb code, it should be noted

that AXThumb code generated at compile time could potentially improve the performance further. There are 2 primary reasons for performance improvement. One, as a result of using AX instructions, registers get freed, allowing the register allocator to take advantage of more free registers. The allocation would occur after instruction selection. Since AX instructions enable the use of higher order registers (`r8-r12`), the register allocator would have to treat AXThumb pairs as a special case (like `mov` instructions in existing Thumb code - the Thumb `mov` instruction can access higher order registers). Two, the instruction scheduler could schedule instructions so as to increase the number of AXThumb pairs generated. Thus, our postpass approach provides a baseline for performance improvement using AX instructions. The algorithms for each of the three phases in the postpass approach, along with code examples, are described in detail next.

3.1.1 Phase 1 - Predicated Code

The code segment shown below illustrates how Thumb code can be predicated using the `setpred` instruction.

Thumb Code
(1) cmp r3, #0
(2) beq (6)
(3) sub r6, r1
(4) sub r5, r2
(5) b (8)
(6) add r6, r1
(7) add r5, r2
(8) mov r3, r9

AXThumb Code
(1) cmp r3, #0
(2) setpred EQ, #2
(3) add r6, r1
(4) sub r6, r1
(5) add r5, r1
(6) sub r5, r2
(7) mov r3, r9

The original Thumb code has to execute explicit branch instructions to achieve conditional execution, choosing between the subtract and add operations. Using the `setpred` instruction we can avoid this explicit branching. This instruction specifies two things. First it specifies the *condition* involved in predication (e.g., `eq`, `ne` etc.).

```
input  : A CFG for a function
output: A modified CFG with 'set'predicated code
for all siblings (n1, n2) in the BFS Traversal of the CFG do
    /* Check for a hammock in the CFG */
    PredEQ = SuccEQ = FALSE;
    if numPreds (n1) == numPreds (n2) == 1 then
        if Pred (n1) == Pred (n2) then
            PredEQ = TRUE;
        end
    end
    if numSuccs (n1) == numSuccs (n2) == 1 then
        if Succ (n1) == Succ (n2) then
            SuccEQ = TRUE;
        end
    end
    /* SetPredicate if hammock found */
    if SuccEQ and PredEQ then
        DeleteLastIns( Pred(n1));
        InsertIns( Pred( n1), setpred, cond );
        for each pair of instructions in1, in2 from n1 and n2 do
            InsertIns( Pred(n1 ), in1 );
            InsertIns( Pred(n1 ), in2 );
        end
        MergeBB(Pred(n1), Succ(n1));
        DeleteBB(n1);
        DeleteBB(n2);
    end
end
```

Algorithm 1: SetPredicate

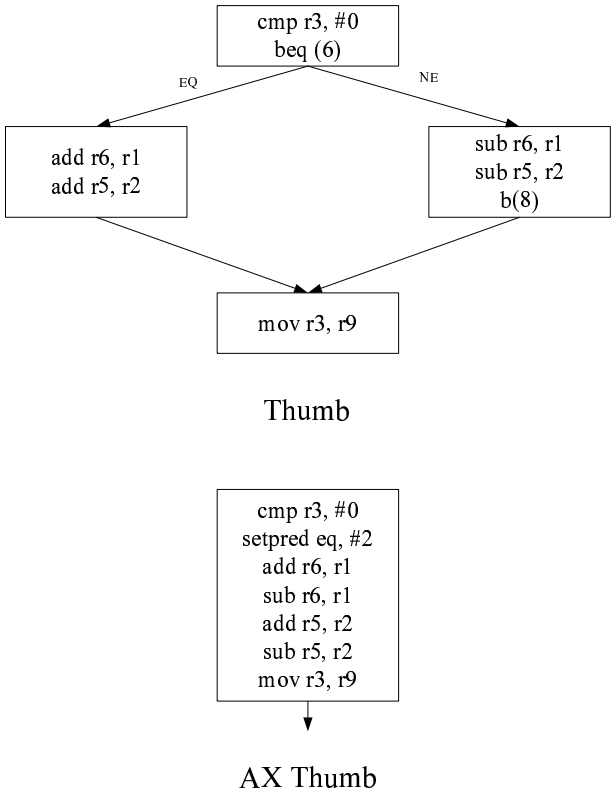


FIGURE 3.1. Predication

Second it specifies the *count* of predicated instruction pairs that follow. Following the `setpred` instruction are pairs of Thumb instructions – the number of such pairs is equal to *count*. If the *condition* is true, the first instruction in each pair is executed; otherwise the second instruction each pair is executed.

The examples shown above is the same as the one described in Section 2.2.2. Although each `setpred` instruction can only predicate upto 8 pairs of instructions, longer blocks of code can be predicated by multiple `setpred` instructions with the same condition for each portion of the large block.

This method of predication is more effective than ARM predication because, in the case of ARM, `nops` are issued for predicated instructions whose condition is not satisfied. Remember, in the case of ARM, every fetch only fetches one 32-bit instructions. Hence, when the predicate is not satisfied, the instruction fetched is not

executed and that cycle is wasted. In the case of Thumb, since two 16-bit instructions from both paths are available, the one that satisfies the predicate is executed while the other is discarded. However this form of predication can be applied only to simple single branch hammocks (or diamond shapes in the `CFG`) corresponding to a simple `if-then-else` construct. Hence, the algorithm described below, first detects such branch hammocks in the `CFG` for the function, then interleaves the instructions from the two branches, merging them with the parent basic block. We consider pairs of sibling nodes during a Breadth-First Traversal of the `CFG` for hammock detection. A hammock is detected when (i) the predecessor of both siblings is the same, (ii) there is exactly one predecessor (iii) and both siblings have the same successor. Once a hammock is detected, it is predicated by inserting a `setpred` instead of the branch instruction and interleaving the code from the two branches as shown in Algorithm 1. The `CFGs` for the code example described above, before and after the transformation are shown in Figure 3.1.

3.1.2 Phase 2 - Peephole Optimizations

The code segment shown next illustrates the general case for AX Transformations which captures the majority of AX instructions. This example uses the `setshift` and `setsource` AX instructions. The `setshift` instruction specifies the type and amount of the shift needed by the following instruction. The `setsource` instruction specifies the high register needed as the source for the following instruction. While the Thumb code requires the execution of five instructions, the AXThumb code only executes three instructions.

Thumb Code
(1) mov r2, r5
(2) lsl r4, r2, #2
(3) mov r3, r9
(4) sub r1, r4
(5) ldr r5, [r3, #100]

AXThumb Code
(1) mov r2, r5
(2,4) setshift lsl #2 sub r1, r2
(3,5) setsource high r9 ldr r5, [-,#100]

input : Basic Block DAG D with nodes numbered according to the topological order and register liveness information

output: Basic Block DAG D with Coalesced Nodes to indicate AXThumb instruction pairs

```
for each n ϵ nodes in BFS order of D do
    for each p ϵ Pred(n) do
        Let dependence between n and p be due to register r.
        if r is not live following instructions (n,p) then
            /* Check if nodes n and p are coalescable */
            if CandidateAXPair(n,p) then
                G ← ∅
                G ← Coalesce(n,p)
                /* Check if coalesced Graph is a DAG */
                isDAG = TRUE
                for each e ϵ edges in G do
                    if Source(e) > Destination(e) then
                        isDAG = FALSE
                    end
                end
                if isDAG then
                    D ← G
                end
            end
        end
    end
end
```

Algorithm 2: DAG Coalescing for generic AX instructions

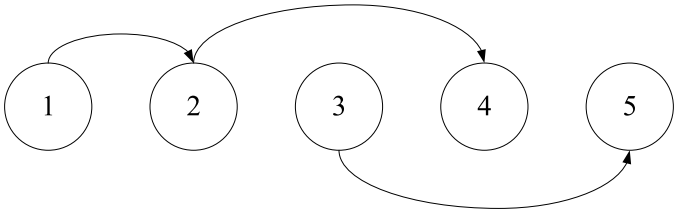
Since these transformations are local to a basic block, the algorithm shown in Algorithm 2 uses the Basic Block dependence DAG as its input. Since AXThumb pairs replace dependent Thumb instructions, it is sufficient to examine adjacent nodes along a path in the DAG. We traverse the DAG in Breadth-First Order and examine

each node with its predecessor. AXThumb pairs have to be instructions adjacent to each other in the instruction schedule. While replacing Thumb pairs with equivalent AXThumb pairs, in order to ensure that this property is maintained, we coalesce the nodes of the candidate Thumb pairs into one node representing the AXThumb pair. However to maintain the acyclic property of the DAG, we have to ensure that this coalescing of candidate Thumb instructions does not introduce a cycle. The nodes in the DAG are numbered according to the topological sorted order of the instruction schedule. By checking for back edges from higher numbered nodes to lower numbered nodes during coalescing we make sure that the acyclic property is maintained. The final instruction schedule is the ordering of nodes according to increasing node id where for coalesced nodes, the node id is the id of the first instruction in the node.

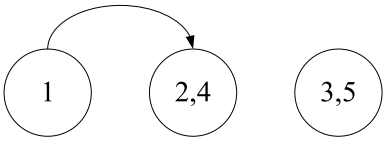
For our example, instructions 3 and 5 are candidates and instructions 2 and 4 are candidates. The `CandidateAXPair` function takes in two Thumb instructions and checks to see if they are candidates for replacement. This involves a liveness check. Using liveness information, in our example one can say that register r4, in instruction 2, is a temporary register. Since the two dependent instructions (subtract and shift) can be replaced using a `setshift` instruction and register r4 is not live after instruction 3, the `CandidateAXPair` function returns the AXThumb pair that could replace instructions 2 and 4. Since coalescing nodes 2 and 4 does not introduce a cycle, the replacement is legal. The algorithm for phase 2 is shown in Algorithm 2 and the DAG for our example, before and after the transformation is shown in Figure 3.2.

3.1.3 Phase 3 - Function Prologues and Epilogues

The third phase handles the specific case of the `setallhigh` instruction, where a whole sequence of Thumb instructions is converted to an AXThumb pair. The code segment shown next illustrates the need for a `setallhigh` instruction. Since only low registers can be accessed in Thumb state, the saving and restoring of context at function



(a) Thumb



(b) AX Thumb

FIGURE 3.2. Phase 2

boundaries results in the use of extra move instructions. In the example above, first the low registers are pushed onto the stack, the high registers are then moved to the low registers before they are pushed onto the stack. Using the `setallhigh` instruction we can avoid the extra moves, indicating that the next instruction accesses high registers.

Thumb Code
`(1) push [r4, r5, r6, r7]`
`(2) mov r4, r8`
`(3) mov r5, r9`
`(4) mov r6, r10`
`(5) mov r7, r11`
`(6) push [r4, r5, r6, r7]`

AXThumb Code	
`(1)`	`push [r4, r5, r6, r7]`
`(2,3)`	setallhigh
	`push [r4, r5, r6, r7]`

This transformation, like phase 2, is local to a basic block and uses the basic block DAG as its input. The algorithm detects such sequences during a Breadth-First traversal of the DAG. The dependence in the DAG is between the push instructions and the move instructions as shown in Figure 3.3. The move instructions are siblings

```
input  : Basic Block DAGs (with nodes in the topological sorted order of the
         instruction schedule) for the basic block predecessors of the exit node and
         successors of the entry node in the CFG and register liveness information
output: Reduced Basic Blocks with setallhigh AX instructions
for each DAG D ϵ set of basic blocks B do
   for each n ϵ BFS order of nodes in D do
      if PushOrPopListLo(n) then
         /* Check for the replaceable mov instructions */
         isReplacable = TRUE
         for each m ϵ Succ(n) do
            Let r be the destination register in m.
            if r is not live following Succ(m) then
               if not movLoHi(m) |
               not PushOrPopListHi(Succ(m)) | numSuccs(m) ≠ 1 then
                  isReplacable = FALSE
               end
            end
         end
         /* Remove MOVs and insert a setallhigh */
         if isReplacable then
            for each m ϵ Succ(n) do
               Save ← Succ(m)
               Remove(m)
            end
            Succ(n) ← Save
            SettoLo(Save)
            Coalesce(setallhigh, Succ(n))
         end
      end
   end
end
```

Algorithm 3: DAG Coalescing for setallhigh AX instructions

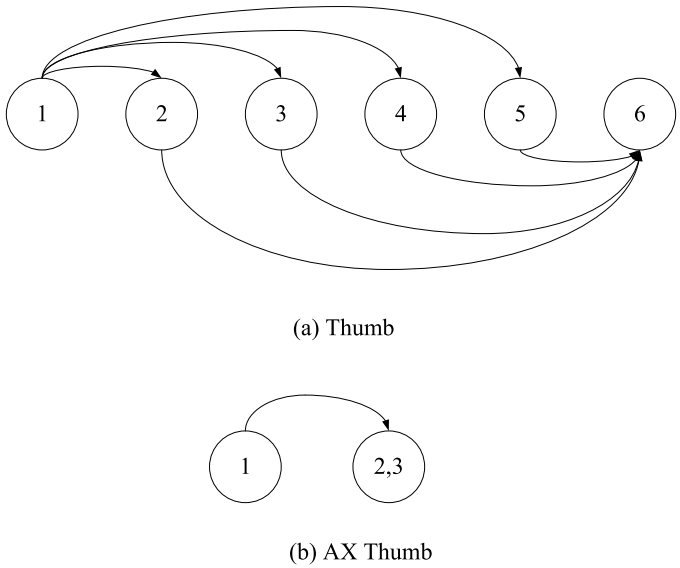


FIGURE 3.3. SetAllHigh AX transformation

with predecessor and successors as the push instructions in the DAG. This condition is checked for as shown in Algorithm 3. The `PushorPopList` functions find instructions that push/pop a list of registers and performs the liveness check on these registers. The `movLoHi` function makes sure the register being used in the mov instruction is in the list of registers in the push/pop instruction encountered before. Once such a pattern is detected all the sibling nodes are replaced with one single node containing the `setallhigh` instruction. This node is then coalesced with the successor node which is the push/pop instruction to ensure that two instructions are adjacent to each other in the instruction schedule.

3.2 Profile Guided Approach for Mixed Code

In this section we provide a description of the Profile Guided Approach for the generation of mixed code [19]. First we describe the instruction support already available in the ARM/Thumb instructions set that allows such mixed code generation. We show why generating mixed code at fine granularity (i.e., for sequences of instructions like

those we described in Section 2.2) results in poorer code. We briefly describe the best heuristic from [19] Heuristic 4 (H4), called PGMC from here on, which generates mixed code at coarser granularity next. We present experimental results comparing AX to PGMC approach along with other experimental results in Section 4. There has been recent work on mixed code generation at compile time, which generates mixed code at a finer granularity than the approach described in [19]. The reader is pointed to [22] for details on this approach.

3.2.1 BX/BLX instructions

The ARM/Thumb ISA supports the Branch with eXchange (BX) and Branch and Link with eXchange instructions. These instructions dictate a change in the state of the processor from the ARM state of execution to the Thumb state or vice versa. When the target register in these instructions `(Rm)` has its 0^{th} bit `(Rm[0])` set the state changes to Thumb otherwise it is in ARM state. These instructions change the Thumb bit of the `CPSR` (Current Program Status Register), indicating the state of the processor.

Using the BX instruction at finer granularity we could generate a mixed binary that targets the specific sequences that AX targets. However this technique is ineffective as we show in Figure 3.4. As we can see from the code transformation shown, when the *longer Thumb sequence* is replaced by a *shorter ARM sequence*, we introduce three additional instructions. Moreover, the alignment of ARM code at word boundary may cause an additional *nop* to be introduced preceding the first `BX` instruction. Hence, for the small sequences that are targeted by AX, this method introduces too much overhead due to the extra instructions leading to a net loss in performance and code size. Therefore, this approach is ineffective when applied at fine granularity. On the other hand if this transformation were applied at coarser granularity, the overhead introduced by the extra instructions can be acceptable. In

the next section we describe a heuristic that carries out mixed code generation at coarser granularity.

Thumb	
`.code 16`	; Thumb instructions follow
...	
<pattern>	
...	

ARM+Thumb	
`.code 16`	; Thumb instructions follow
...	
`.align 2`	; making `bx` word aligned
`bx r15`	; switch to ARM as r15[0] not set
`nop`	; ensure ARM code is word aligned
`.code 32`	; ARM code follows
<ARM code>	; pattern
`orr r15, r15, #1`	; set r15[0]
`bx r15`	; switch to Thumb as r15[0] is set
`.code 16`	; Thumb instructions follow
...	

FIGURE 3.4. Replacing Thumb Sequence by ARM Sequence.

3.2.2 Profile Guided Mixed Code Heuristic (PGMC)

A profile guided approach is used to generate a mixed binary, one that has both ARM and Thumb instructions. This heuristic chooses a coarse granularity where some functions of the binary are ARM instructions while the rest is Thumb. The compiler inserts BX instructions at function boundaries to enable the switch from ARM to Thumb state and vice versa as required. Heuristics based on profiles determine which functions use ARM instructions allowing the placement of BX instructions at the appropriate function boundaries. The basic approach that we take for generating mixed code consists of two steps. First we find the frequently executed functions once using profiling (e.g., using `gprof`). These are functions which take up more than

5% of total execution time. Second we use heuristics for choosing between ARM and Thumb codes for these frequently executed functions. For all other functions, we generate Thumb code. The above approach is based upon the observation that we should use Thumb state whenever possible. For all functions within a module (file of code), we choose the same instruction set. This approach works well because when closely related functions are compiled into mixed code, optimizations across function boundaries are disabled, resulting in a loss in performance.

PGMC uses a combination of instruction counts and code size collected on a per function basis. We use the Thumb code if one of the following conditions hold: (a) the Thumb instruction count is lower than the ARM instruction count; or (b) the Thumb instruction count is higher by no more than `T1`% and the Thumb code size is smaller by at least `T2`%. We choose T1=3 and T2=40 for our experiments. We determined these settings through experimentation across a set of benchmark as discussed in [19]. The idea behind this heuristic is that if the Thumb instruction count for a function is slightly higher than the ARM instruction count, it still may be fine to use Thumb code if it is sufficiently smaller than the ARM code as the smaller size may lead to fewer instruction cache accesses and misses for the Thumb code. Therefore, the net effect may be that the cycle count of Thumb code may not be higher than the cycle count for the ARM code.

3.3 Experiments

The primary goal of our experiments is to determine how much of the performance loss experienced by the use of Thumb code, as opposed to ARM code, can be recovered by using the AX instruction set and instruction coalescing. To carry out this experimentation we implemented the described techniques in our simulation and compilation environment. Then we ran the ARM, Thumb and AXThumb versions of the programs and compared their performance. We describe the experimental setup

followed by a discussion of the results.

Experimental setup A modified version of the Simplescalar-ARM [2] *simulator*, was used for experiments. It simulates the five stage Intel's SA-1 StrongARM pipeline [14] with an 8-entry instruction fetch queue. The I-Cache configuration for this processor are: 16Kb cache size, 32b line size, and 32-way associativity, and miss penalty of 64 cycles (a miss requires going off-chip). The simulator was extended to support both 16-bit and 32-bit modes, the Thumb instruction set and the system call conventions followed in the `newlib` c library. This is a lightweight C library used on embedded platforms that does not provide explicit network, I/O and other functionality typically found in libraries such as `glibc`. CACTI [30] was used to model I-Cache Energy. The `xscale-elf gcc version 2.9` compiler used was built to create a version that supports generation of ARM, Thumb as well as mixed ARM and Thumb code. Code size being a critical constraint, all programs were compiled at -O2 level of optimization, since at higher levels code size increasing optimizations such as function inlining and loop unrolling are enabled. The *benchmarks* used are taken from the `Mediabench` [21], `Commbench` [36] and `NetBench` [10] suites as they are representative of a class of applications important for the embedded domain. The benchmark programs used do not require functionality not present in `newlib`. A brief description of the benchmarks is given in Table 3.1. All experiments used a single workload for each benchmark program.

3.3.1 Performance of AXThumb

Instruction Counts The use of AX instructions reduces the dynamic instruction count of 16-bit code by 0.4% to 32%. Figure 3.5 shows this reduction normalized with the counts for 32-bit ARM code. The difference in instruction count between ARM and Thumb code is between 3% and 98%. Using AX instructions we reduce the performance gap between 32-bit and 16-bit code. For cases such as `crc` and

TABLE 3.1. Benchmark Description

Name	Description
`rtr`	Routing Lookup Algorithm
`crc`	Cyclic Redundancy Check Algorithm
`adpcm`	Adaptive Differential pulse code modulation (encode/decode)
`pegwit`	Elliptical Curve Public key Encryption Algorithm
`frag`	IP packet header fragmentation
`reed`	Reed Solomon Forward Error Correction Algorithm
`drr`	Deficit Round Robin Scheduling

`adpcm` where there is substantial difference between ARM and Thumb code, we see improvements between 25% and 30% bridging the performance gap between ARM and Thumb by one third in the case of `crc` and more than one half in the case of `adpcm`. For cases such as `drr` where Thumb code is not much worse than ARM code (3%), we see little improvement using AX instructions. In the other cases we see an improvement over Thumb code of about 10% on an average. The difference in the instruction counts between ARM and Thumb code indicates the room for possible improvement of 16-bit code due to constraints present in 16-bit code. Using AX instructions we are able to considerably bridge this gap between 32-bit and 16-bit code.

Cycle Counts Figure 3.6 shows the cycle count data for Thumb and AXThumb code relative to the ARM code. The use of AX instructions gives varying cycle count changes between -0.2% and 20% compared to Thumb code. We see reduction of 15% to 20% in cycle counts for `crc` and `adpcm` compared to the Thumb making the reducing the difference between ARM and Thumb by half in the case of `crc` and about 66% with the `adpcm` programs. In the other 3 cases where Thumb cycle counts are higher than ARM, viz. `frag` `reed.encode`, `reed.decode`, and `rtr`, we see that there is a moderate reduction in cycle counts compared to Thumb. However the

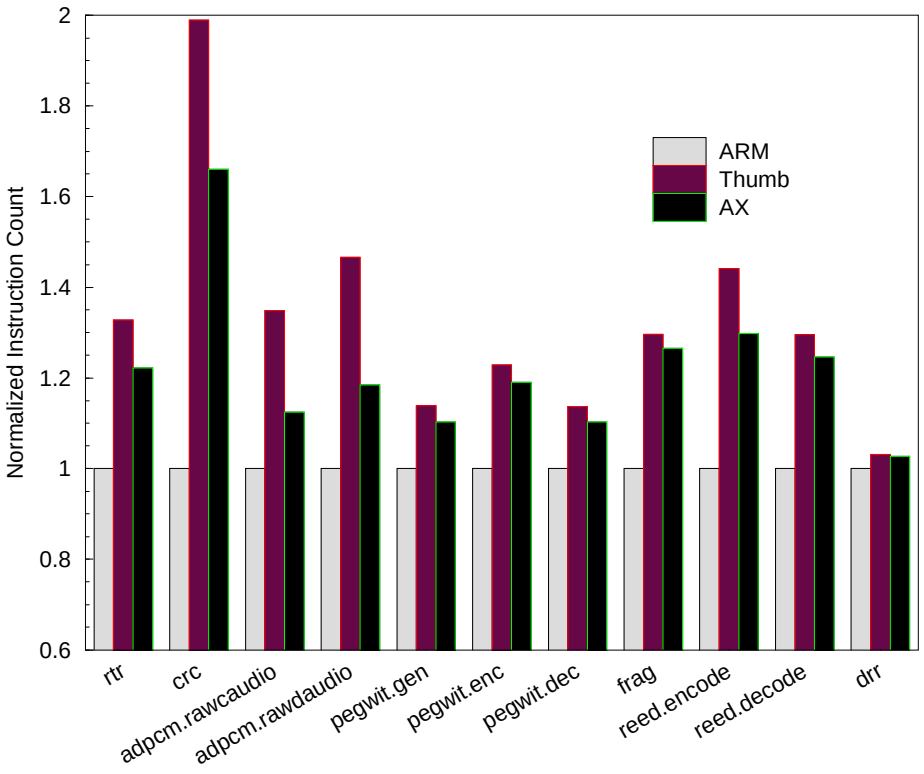


FIGURE 3.5. Normalized Instruction Counts

difference between the ARM and Thumb codes itself being moderate, in the cases of `rtr` and `reed.encode`, AXThumb code gives a lower cycle count compared to even ARM code. The improved I-cache behavior of the Thumb and AXThumb codes compared to ARM code makes this possible. In the other cases, where Thumb code already outperforms ARM code we see little improvement as there is little scope for the use of AX instructions.

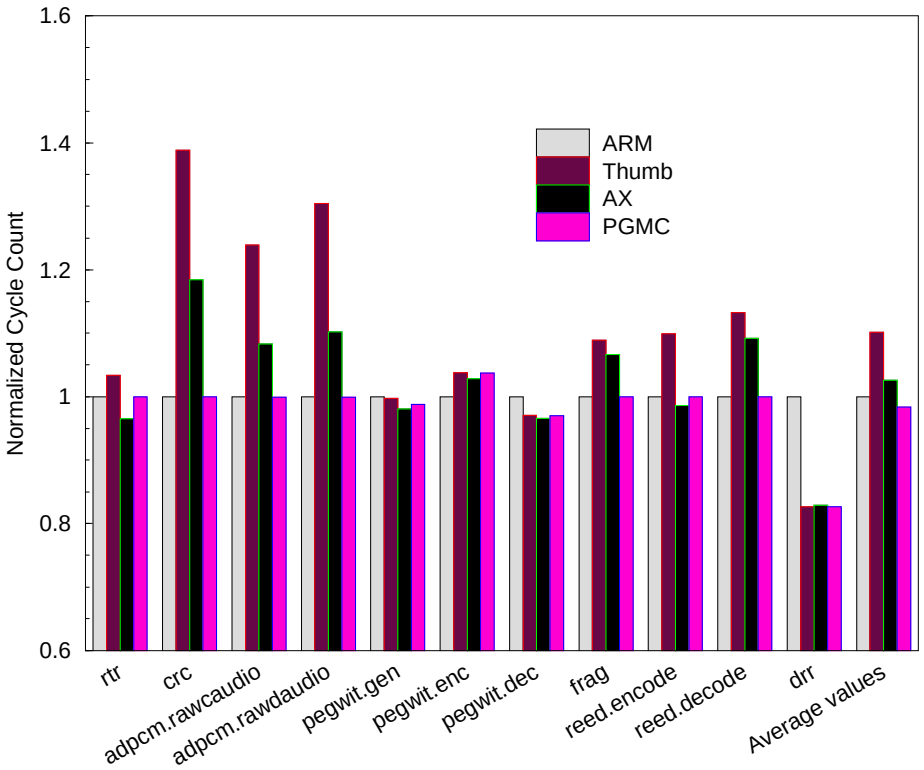


FIGURE 3.6. Normalized Cycle Counts

Code Size and I-Cache Energy The code sizes of Thumb and AXThumb are almost identical. This is because in all cases where AXThumb instruction replace Thumb instructions, the size is only decreased if at all changed. The decrease occurs due to the introduction of `setallhigh` or `setpred` instructions as mentioned before. In all other cases the size does not change. The code sizes relative to ARM are shown in Figure 3.7. Figure 3.8 shows the I-cache energy for Thumb and AXThumb codes

relative to ARM code. In the three cases where Thumb has higher I-cache energy viz. `crc` and the two `adpcm` programs, we see that AXThumb reduces the I-cache energy making them almost as small as ARM. In the other cases we see AX always has lower I-cache energy compared to Thumb, making it even better compared to ARM. Lower I-cache energy results from fewer fetches from the I-cache. Fewer fetches could result from code size reducing AX transformations such as, setpred, setallhigh and negative immediate offset examples shown in section 2.2. Additionally, the number fetches into the instruction queue depends on the utilization of the queue. AXThumb consumes instructions at a faster rate from the instruction queue compared to Thumb, filling up the queue slower compared to Thumb. Hence, on taken branches when the queue is flushed, there are fewer instructions that are flushed, which account for the extra fetches performed by Thumb. Since the instruction count is reduced, energy spent during instruction execution, in other parts of the processor is also reduced. The addition of the AX processor in the decode stage is a very small increase in energy spent since the operations of the AX processor are very simple involving detection of the AX opcode and setting the status if the instruction is an AX instruction. However, this small amount of energy is spent every cycle. The I-cache consumes a significant portion the total energy (upto 25% in some implementations [32]) while the decode stages consume little energy. Hence, savings in I-cache energy translate significant overall energy savings. Thus, while more energy is spent in the decode stage, there is a significant savings from the I-cache. An accurate estimation of energy would require an energy model for all parts of the processor during our simulation. Currently, our infrastructure only models I-cache energy behavior.

Usage of AX instructions In Table 3.2 we show a weighted distribution of the AX instructions executed by each benchmark. Each benchmark uses a different set of AX instructions and all AX instructions have been used by at least two benchmarks. Instructions that made an impact in almost all benchmarks were `setsbit`, `setshift`,

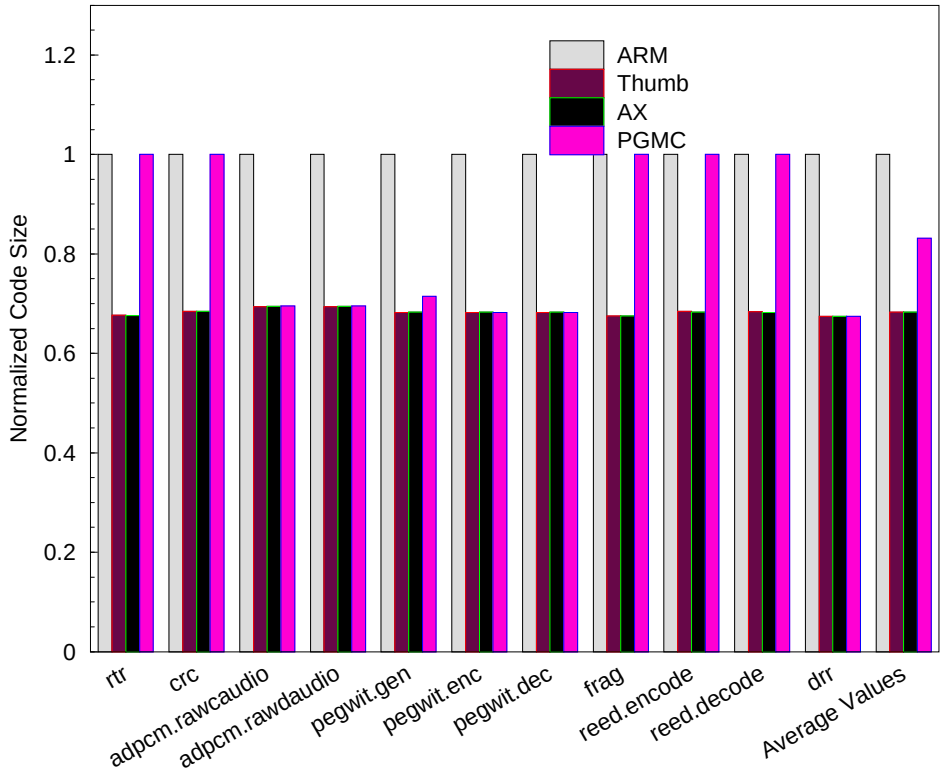


FIGURE 3.7. Normalized Code Size

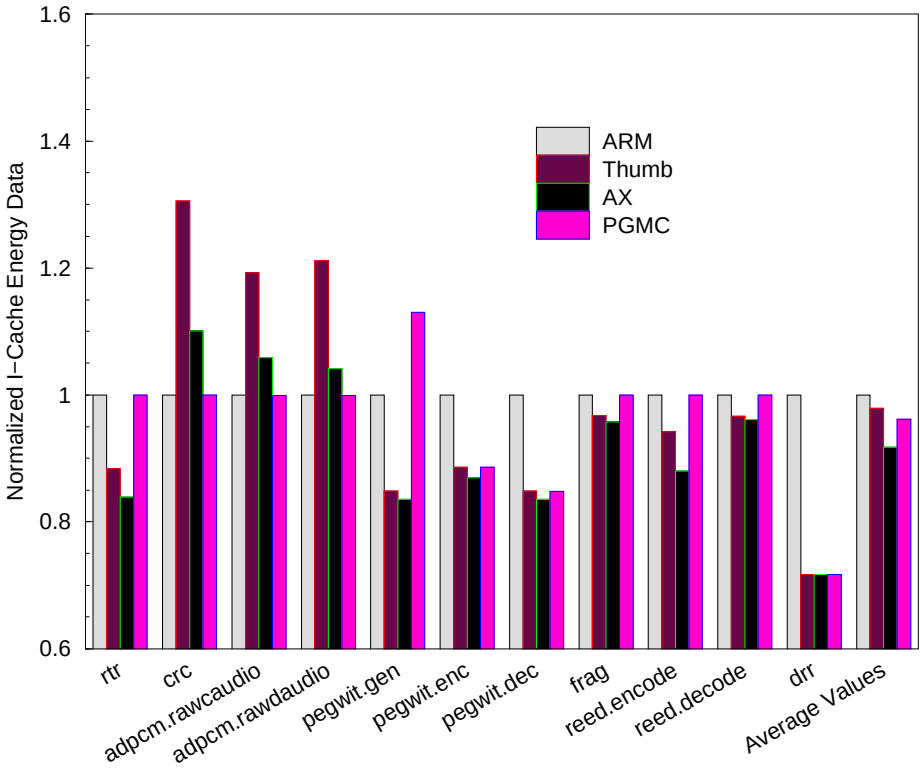


FIGURE 3.8. Normalized I-Cache Energy

`setsource` and `setthird`. Predication was found to be useful only in `adpcm` as in other benchmarks small branch hammocks capable of being predicated were not found. In `crc`, a small set of `setsbit` instructions in the hotspots of the code gave very good performance improvement. `drr` had little opportunity for insertion of AX instructions resulting in the use of a few `setsbit` instructions which did not give much of an improvement. The use of `setallhigh` in `rtr` resulted in smaller code as a result of removing unnecessary moves, which was also the reason for reduced instruction count.

3.3.2 Comparison with Profile Guided Mixed Code

Cycle Counts Figure 3.6 also shows the cycle counts for PGMC normalized with ARM cycle counts. `crc` is the only benchmark where AX cycle counts are considerably more than PGMC. For most of the other benchmark the AX and PGMC counts are very close. In some cases like `adpcm`, `frag` and `reed.decode` PGMC has lower cycle counts; while in other cases like `rtr`, `pegwit` and `reed.encode` AX has lower cycle counts. In some cases for PGMC like `rtr`, `crc` and `adpcm` the heuristic chooses all modules to be compiled into ARM code. In the case of `drr` PGMC chooses to compile all modules into Thumb code. The cycle counts for these benchmarks reflect these decisions.

Code Size Figure 3.7 also shows the code size for PGMC normalized with respect to the ARM code sizes. We see that for quite a few benchmarks, PGMC is significantly worse than AX. Also notice how AX always has smaller code size compared to PGMC. As indicated above the reason for larger code size in PGMC is due to the choice of using only ARM code. The amount of memory required for AX is in general lesser than PGMC.

TABLE 3.2. Usage of Different AX Instructions.

Benchmark	setallhigh	setpred	setsbit	setshift	setsource	setdest	setthird	setimm
rtr	11.77%	0.00%	82.34%	5.88%	0.00%	0.00%	0.00%	0.00%
crc	0.00%	0.00%	0.27%	99.72%	0.00%	0.00%	0.00%	0.00%
adpcm.rawcaudio	0.00%	36.30%	36.30%	14.52%	0.00%	7.26%	0.00%	5.59%
adpcm.rawdaudio	0.00%	34.47%	34.47%	13.79%	3.44%	10.34%	3.44%	0.00%
pegwit.gen	0.17%	0.00%	74.47%	8.48%	5.47%	0.00%	11.39%	0.00%
pegwit.encrypt	0.19%	0.00%	80.22%	5.01%	6.23%	0.00%	8.32%	0.00%
pegwit.decrypt	0.17%	0.00%	74.47%	8.48%	5.47%	0.00%	11.39%	0.00%
frag	4.44%	0.00%	0.00%	6.66%	13.33%	4.44%	66.66%	4.44%
reed.encode	0.01%	0.00%	3.81%	0.00%	68.45%	0.00%	27.71%	0.00%
reed.decode	0.01%	0.00%	1.09%	0.63%	88.29%	0.00%	9.95%	0.00%
drr	0.00%	0.00%	100.00%	0.00%	0.00%	0.00%	0.00%	0.00%

I-Cache Energy Figure 3.8 also shows the I-Cache energy for PGMC normalized with I-cache energy for ARM code. PGMC has higher I-cache energy for all but 3 benchmarks. This is significant in benchmarks like `pegwit.gen` and `rtr`, and less significant in other benchmarks like `reed` and `frag`. In the other 3 programs we notice AX is slightly worse than PGMC.

From the above results we see that AX and PGMC, each have some advantages over the other. PGMC has better performance in general while AX has smaller code size. With the support of more AX type of instructions, one could possibly further improve performance. From an energy perspective, with our current infrastructure, it is hard to estimate accurately which is superior. Instruction coalescing, if carried out with more AX style of instructions, could possibly remove the need to support the 32-bit ISA and still achieve performance of 32-bit code.

3.4 Summary

In this chapter we have seen how the compiler can use the AX instructions described in the previous chapter to effectively improve the performance of Thumb code. The local optimizations were carried out in three phases. The first phase predicates Thumb code using the predication support exposed via the *setpred* instruction. The second phase exploits peephole opportunities which replaces pairs of Thumb instructions with an AX-Thumb pair. The third phase minimizes the call overhead in Thumb programs. Experiments showed improvements in Thumb code were achieved with insignificant increase in code size. A comparison with our prior work on Profile Guided mixed code generation showed that DIC is superior.

Chapter 4
Global Optimization Using DIC

This chapter introduces techniques that exploit Dynamic Instruction Coalescing for more efficient use of the register file. Half the register file is invisible to the compiler while generating 16-bit code. This is the result of register specifiers being 3-bits wide in the 16-bit ISA compared to 4-bits wide in the 32-bit ISA. So the compiler can only allocate half the register file which results in many more memory operations. Our technique exploits these extra registers by sharing encoding between registers and using microarchitectural and compiler support to expose and efficiently exploit these invisible registers.

An AX instruction, setmask, introduces the notion of an active subset of registers. The register file is partitioned into two halves of 8 registers; each register in the low half shares an encoding with a corresponding register in the high half. The setmask instruction is used to activate different subsets of 8 registers from the available 16 registers. The compiler is exposed to all registers as a result and can allocate all of them. After allocation, the compiler inserts setmask instructions to ensure correct switching between register subsets. Like the previous AX instructions we seen, the setmask AX instruction is a zero cycle instruction that does not contribute to the execution time of the program. However, unlike previous AX instructions, setmask AX instructions do not replace existing 16-bit instructions; the setmask AX instruction is an additional instruction. Hence, apart from ensuring correct insertion of AX instructions the compiler also ensures minimal insertion of setmask instructions to avoid code bloat. First we will see how invisible registers are exposed and then we will look at effective ways to exploit them.

4.1 Exposing Invisible Registers

In Thumb instructions the register specifier field is typically 3 bits while it is 4 bits in all ARM instructions. Thus, in Thumb mode the lower half of the register file (i.e., $R0 \cdots R7$) can be freely accessed while the upper half (i.e., $R8 \cdots R15$) of the register file is accessible only by very few instructions (e.g., MOVs).

The dynamic coalescing framework can be used to enable the higher order registers visible to all Thumb instructions. To achieve this goal we take the following approach. We view the register file containing 16 registers as consisting of 8 register pairs – $(r0, r8)$, $(r1, r9)$, $\cdots$ $(r7, r15)$. Only 8 registers are visible at a time such that exactly one register is visible from each pair at any point in time. For each register pair, the register from the pair that is visible at any point in time needs to be set. For this purpose we provide the `SetMask` instruction. This instruction has an 8 bit operand where each bit in the operand specifies the leading bit of the register specifier. If the leading bit for a pair is 0 then the lower order register is visible while if the leading bit is 1 then the higher order register from the pair is visible. Each time in the program when the set of visible registers needs to be changed, a single `SetMask` instruction is executed to achieve this goal.

The execution of the `SetMask` instruction is achieved without adding additional execution cycles using the dynamic coalescing framework. A `SetMask` instruction is processed in the decode stage in parallel with the preceding Thumb instruction in the same manner as other AX instructions are accessed as described in the preceding section. The decode stage saves the bits specified in the `SetMask` and uses these bits to interpret the register specifier bits in future Thumb instructions till the next `SetMask` instruction is encountered. While the above approach greatly reduces the constraints on use of registers by Thumb instructions, one minor constraint still remains – a Thumb instruction cannot simultaneously reference both registers from a register pair. The register allocator must take this constraint into account during register

assignment.

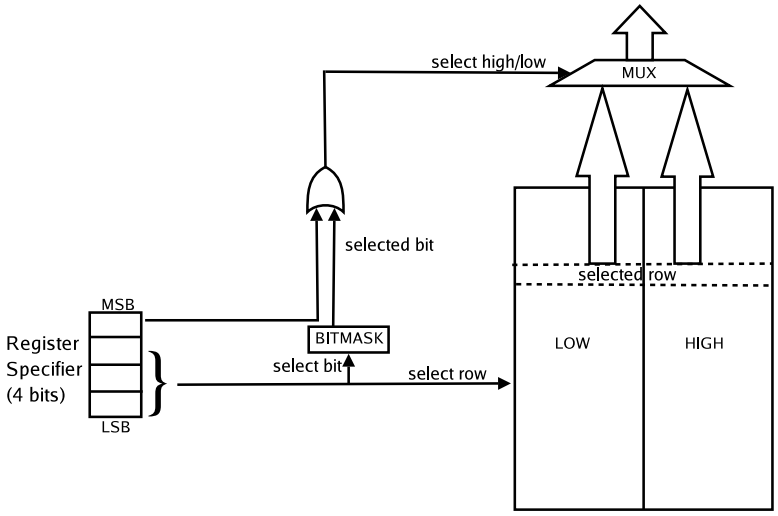


FIGURE 4.1. Register Operand Access

Figure 4.1 shows how register operands use the bitmask set by the `SetMask` instruction. The register file needs to be modified to accommodate register operand access without delays. The register file is organized such that each row contains the high and low registers. The lower 3 bits of the register specifier are used to index the register file as well as the bitmask. In 16-bit state the bit selected from the bitmask is used to select the low or high register contents of the selected row. In 32-bit ARM state the MSB of the register specifier selects the low or high register. Both the bitmask and register file are indexed in parallel to avoid introducing delays during register file access.

Let us consider a simple example of Figure 4.2 that illustrates the advantage of using `SetMask` instruction. In this example, we assume that registers $r0$ and $r5$ are available to hold the values of variables a and c respectively immediately preceding and following the code fragment. The variable t is a temporary which is computed and consumed within the code fragment. Let us assume that other than $r0$ and $r5$ the only register available is $r10$ while all other lower and higher order registers are already occupied by other variables. Under these assumptions we show the generated

code sequences for ARM, Thumb, and AXThumb. As we can see, in case of ARM only two instructions are generated where $r10$ is used for the temporary t. When `SetMask` instruction is used, AXThumb code generated is similar to the ARM code except that the `SetMask` is used so that make $r10$ visible. The number of cycles it takes to execute the ARM and AXThumb codes is the same as the `SetMask` instruction does not cost extra cycles. Now we finally look at Thumb code in which case $r10$ is used to spill the value in $r0$ since $r10$ cannot be directly accessed by the ADD. As we can see, spill code must be generated that causes extra load and store instructions to be generated – we have observed these effects in the Thumb code generated by the `gcc` compiler.

[a in r0; c in r5;] [t - temporary]			

t = c + 5
a = a + t
	mov r10, r0
	add r0, r5, #5
	st r5, < *addr* >
	mov r5, r10
	add r0, r0, r5
	ld r5, < *addr* >
		add r10, r5, #5
		add r0, r0, r10
			setmask 0x04
			add r2, r5, #5
			add r0, r0, r2

(a) IR	(b) Thumb	(c) ARM	(d) AX Thumb

FIGURE 4.2. Use of `SetMask`.

4.2 Exploiting Exposed Registers

It is clear that by using `SetMask` instructions, the execution time of code can be improved in comparison to Thumb code. However, there is another important issue that we must deal with. We would like to ensure that the code size of AXThumb code is also small. While extra `SetMask` instructions are needed, by using higher order registers some of the spill code is eliminated. A *naive approach* for introducing `SetMask` instructions may be to add a `SetMask` instruction before and after each reference to a higher order register. However, this approach adds too many instructions causing a significant increase in the code size as shown in Figure 4.3. Therefore in this section we develop compiler algorithms for carefully introducing `SetMask` instructions.

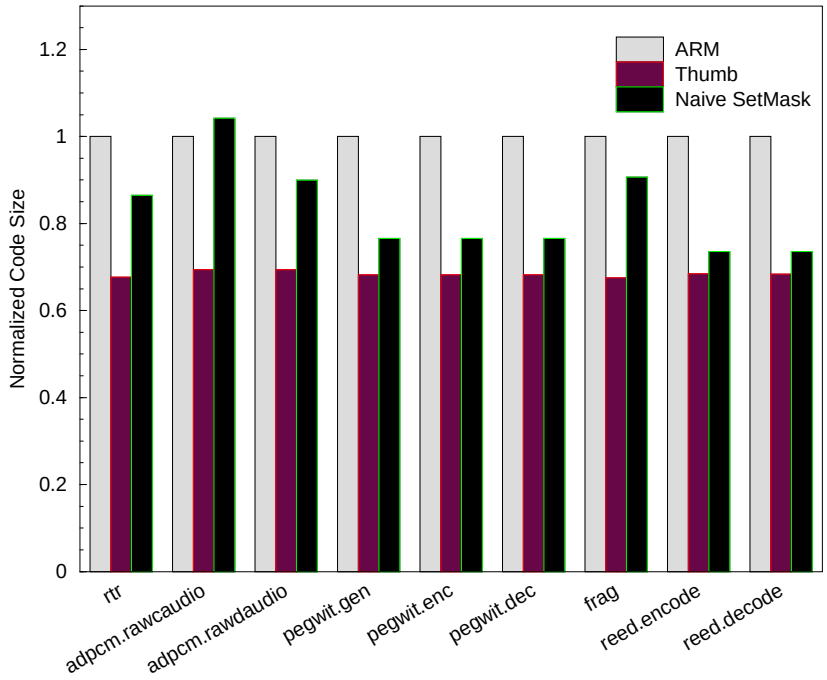


FIGURE 4.3. Normalized Code Size

In the rest of this chapter we will use the following notation. $(R, \bar{R})$ refers to a register pair. While a `SetMask` instruction selects a register to make visible from each pair, when we use the notation `SetMask` R we will be referring to a `SetMask` instruction which makes R visible while keeping the visible registers from remaining pairs the same.

We assume that the Thumb register allocation is performed so that it uses all registers. After register allocation has been performed, the resulting code is examined and `SetMask` instructions are appropriately introduced. The algorithm for `SetMask` placement consists of three main steps. The first step separately determines *initial placement points* for `SetMask` R and `SetMask` $\bar{R}$ instructions corresponding to each register pair $(R, \bar{R})$. The second step finds the *placement ranges* for each `SetMask` R and `SetMask` $\bar{R}$ instruction which represent all points where an instruction can be placed. The third and final step *coalesces* multiple `SetMask` instructions referring to different register pairs into a single `SetMask` instruction that simultaneously changes

the visibility of multiple registers and determines the `final placement` of `SetMask` instructions. The placement ranges identified in the second step are used to identify final placement points which enable maximal coalescing so that fewer `SetMask` instructions are introduced. Next we discuss these steps in detail.

Forward Availability

Initialize:

$$(R,\bar{R})Avail_n(s) := (R,\bar{R})Avail_x(s) := \phi$$

Solve:

$$(R,\bar{R})Avail_n(s) := \bigcap_{p \in Pred(s)} (R,\bar{R})Avail_x(p)$$

$$(R,\bar{R})Avail_x(s) := \begin{cases} (R,\bar{R})Avail_n(s) & R \notin Ref(s) \ \wedge \ \bar{R} \notin Ref(s) \\ \{R\} & R \in Ref(s) \\ \{\bar{R}\} & \bar{R} \in Ref(s) \end{cases}$$

Initial Placement Points

$$(R,\bar{R})Initial_n(s) := \begin{cases} \{SetMask\ R\} & R \in Ref(s) \ \wedge \ R \notin (R,\bar{R})Avail_n(s) \\ \{SetMask\ \bar{R}\} & \bar{R} \in Ref(s) \ \wedge \ \bar{R} \notin (R,\bar{R})Avail_n(s) \\ \phi & otherwise \end{cases}$$

$$(R,\bar{R})Initial_x(s) := \phi$$

FIGURE 4.4. Step 1: Initial Placement Points Determination.

4.2.1 Initial Placement Points

Given a register pair $(R,\bar{R})$, a simple way of introducing a `SetMask` instructions for this register pair is as follows. Immediately preceding an instruction that refers to R we can introduce a `SetMask` R instruction which makes R visible while immediately preceding an instruction that refers to $\bar{R}$ we introduce a `SetMask` $\bar{R}$ instruction which makes $\bar{R}$ visible. Clearly this approach will introduce a lot a `SetMask` instructions – preceding each instruction as many `SetMask` instructions will be introduced as the

number of registers referenced by the instruction. The goal of our algorithm (all three steps) is to reduce this number.

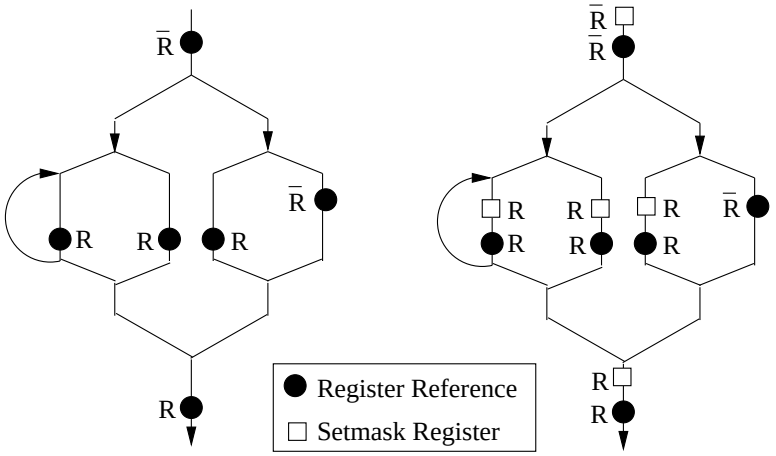


FIGURE 4.5. Initial Placement Points.

In this first step we separately consider each register pair $(R, \bar{R})$ and simply try to eliminate unnecessary `SetMask` R and `SetMask` $\bar{R}$ instructions that are introduced by the above simple strategy. The basic idea of this step is illustrated in Figure 4.5. The first figure shows all references to R and $\bar{R}$ which are marked by ovals. The second figure shows the initial placement points of `SetMask` R and `SetMask` $\bar{R}$ instructions which are marked by squares. As we can see, there is no `SetMask` $\bar{R}$ introduced preceding the second $\bar{R}$ reference. This is because once we introduce `SetMask` $\bar{R}$ before the first $\bar{R}$ reference, the one before the second $\bar{R}$ becomes redundant.

The determination of initial placement points for a given $(R, \bar{R})$ pair is made using the analysis shown in Figure 4.4. Forward *must availability* analysis is used to determine whether or not there is a need to introduce a `SetMask` $R/\bar{R}$ instruction preceding a $R/\bar{R}$ reference. If `SetMask` $R/\bar{R}$ instruction has been executed along all paths prior to reach a reference $R/\bar{R}$ such that $R/\bar{R}$ is already visible, there is no need the introduce a `SetMask` $R/\bar{R}$ instruction before this reference. The $(R, \bar{R})Avail$ sets are computed for all program points as shown in Figure 4.4 and then using this information the initial placement points are determined in form of $(R, \bar{R})Initial$ sets.

Placement Ranges

Initialize:

$$(R, \bar{R})Range_{n/x}(s) := (R, \bar{R})Initial_{n/x}(s)$$

Solve:

$$(R, \bar{R})Range_x(s) := \bigcap_{q \in Succ(s)} (R, \bar{R})Range_n(q)$$

$$(R, \bar{R})Range_n(s) := (R, \bar{R})Range_x(s) - (R, \bar{R})Kill(s)$$

where,

$$(R, \bar{R})Kill(s) := Kill(s, R) \cup (Kill(s, \bar{R})$$

$$Kill(s, R) := \begin{cases} \{SetMask\ R\} & \bar{R} \in Ref(s)\ \vee\ R \in (R, \bar{R})Avail_n(s) \\ \phi & otherwise \end{cases}$$

$$Kill(s, \bar{R}) := \begin{cases} \{SetMask\ \bar{R}\} & R \in Ref(s)\ \vee\ \bar{R} \in (R, \bar{R})Avail_n(s) \\ \phi & otherwise \end{cases}$$

FIGURE 4.6. Propagating Initial Placement Points Backwards to Build Placement Ranges.

4.2.2 Placement Ranges

The previous step determined the latest points at which `SetMask` $R/\bar{R}$ instructions can be placed as they are placed just immediately before $R/\bar{R}$ references. However, we do not simply place them at the initial placement points. This is because we would like to combine instructions corresponding to different register pairs and place them as part of a single `SetMask` instruction – after all, the `SetMask` instruction being supported allows us to simultaneously affect the visibility of registers within each register pair. We would like to identify program points where multiple initial instructions can be coalesced and placed. Thus, starting from the initial point placements, we perform analysis that determines all the places where the instructions can be placed, i.e. we expand initial placement points into *placement ranges*.

The placement range *corresponding to a* `SetMask` $R/\bar{R}$ *instruction is a contiguous portion of the control flow graph such that the instruction can be placed at any point in the range. Each range has distinguishing* earliest points *and* latest points. *An*

earliest (latest) point belonging to a placement range is a point such that none of its predecessor (successor) points belong to the placement range.

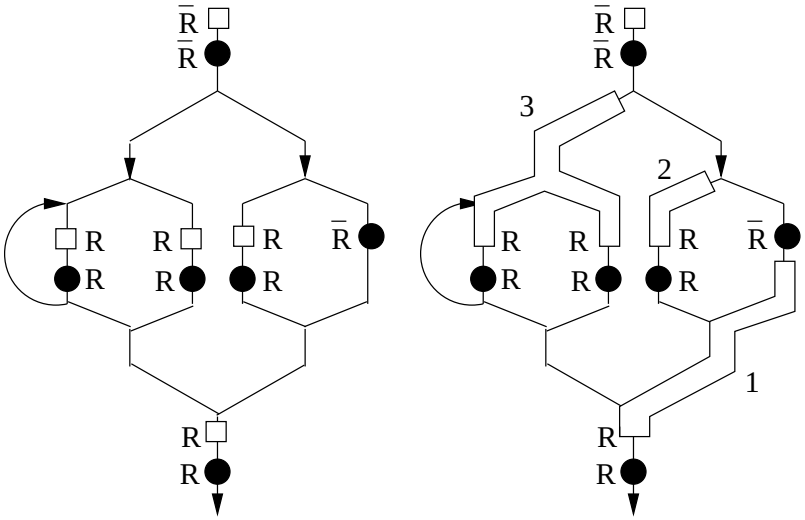


FIGURE 4.7. Placement Ranges.

Before we present the detailed analysis for identifying placement ranges, we illustrate this process by continuing with the example we used to illustrate initial placement point identification. In Figure 4.7, placement ranges corresponding to the initial placement points are shown. Lets look at the three ranges marked 1, 2, and 3 in detail as they demonstrate the various cases that we must take into account in designing the analysis:

- The range marked 1 indicates that while the latest point for placement of `SetMask` R instruction was immediately prior to the reference to R, its earliest placement point is the point that immediately follows the reference to $\bar{R}$. In fact, this instruction can be safely placed at any point along the range that extends from the earliest point to the latest point. There is no need to place this instruction along the paths that merge into this range because `SetMask` R is already available along those paths.

- The range marked 2 extends to the point just before the split point. The earliest point cannot extend above the split point because if `SetMask` R is placed above the split point, the availability of `SetMask` $\bar{R}$ preceding the first $\bar{R}$ at the second $\bar{R}$ will be disrupted.

- The range marked 3 is interesting in that it includes two distinct initial (latest) placement points of `SetMask` R. However, it results in a single earliest point. In other words, if we place the instruction at the earliest point we need one instruction but if we place it at the latest points we need two instructions. Note that in ranges marked 1 and 2 there was a single latest point and single earliest point.

Based upon the illustration above, we are now ready to state the conditions under which a placement point of `SetMask` R can continue to expand into a live range through backward propagation. We can continue to extend the range of `SetMask` R backwards along program points as long as a reference to $\bar{R}$ is not encountered and a point is not reached where `SetMask` R is already available according to the analysis carried out in Step 1 of our algorithm. Based upon these conditions we define the $Kill$ sets used to stop propagation. When a split point is reached, we use the intersection operator to decide whether to continue propagation. As we can see, the intersection operation will correctly prevent and allow propagation above split points for formations of ranges 2 and 3 respectively in our example. The detailed analysis equations are given in Figure 4.6. The results of this analysis are interpreted as follows – for each program point that belongs to a range for `SetMask` $R/\bar{R}$, the set $(R, \bar{R})Range$ is set to {`SetMask` $R/\bar{R}$}; otherwise it is set to empty. As we can see, the results of analysis of Step 1 play a crucial role in Step 2. First, the initialization of $(R, \bar{R})Range$ values at program points is based upon the initial points identified in Step 1. Second, the $Kill$ sets needed during propagation require the use of $(R, \bar{R})Avail$ information also computed during Step 1. The backward propagation

conditions identified above are used by the $Kill$ sets and intersection operator is used at split points.

4.2.3 Coalescing and Final Placement

Now we know the placement ranges of all `SetMask` instructions for each register pair. The goal of this step is to choose final placement points of `SetMask` instructions in a way that enables coalescing of `SetMask` instructions belonging to different register pairs. Such coalescing will reduce the number of instructions introduced. The continuation of our example illustrates the choices. In Figure 4.8, the figure on the left considers the situation in which `SetMask` instructions are only needed for $(R, \bar{R})$. Therefore no coalescing opportunities exist and the choice as shown is made. Now let us assume that there are points at which `SetMask` instructions for $R1$, $R2$ and $R3$ are definitely needed at the points shown in the figure on the right. The `SetMask` instructions for R can be coalesced with them resulting in the placement shown.

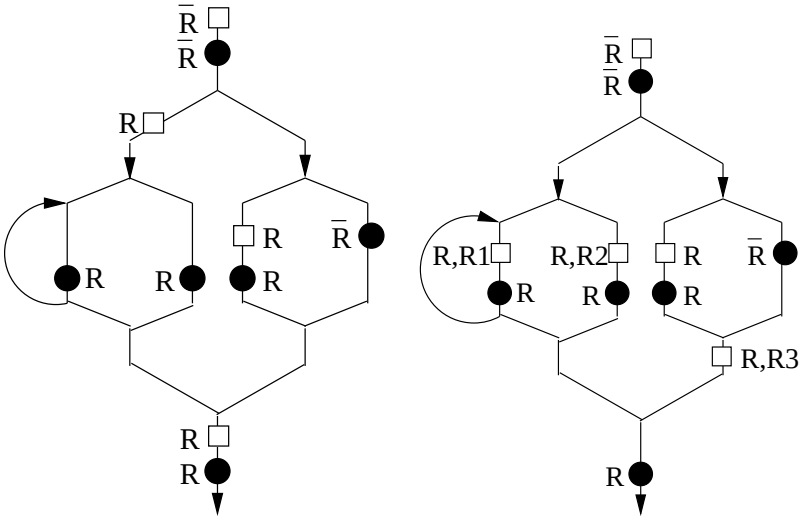


FIGURE 4.8. Final Placement Points.

In order to develop an algorithm for this final step we make the following observation. Given a placement range for say R, depending upon the selection of placements

points the number of `SetMask` R instructions needed to handle this range can vary (e.g., for range marked 3 of Figure 4.7, we may need one or two instructions depending upon the placement points). Moreover, when all register pairs are considered the best overall choice for R need not be the one which requires minimum number of `SetMask` R instructions. This is because the additional cost of placing `SetMask` R instructions depends upon whether or not they can be coalesced with similar instructions for other register pairs.

To explore coalescing opportunities, we develop a formulation of placement point selection where the placement point decisions for all register pairs can be made simultaneously. The key first step in this formulation is the decomposition of placement ranges into *placement paths*.

We define a placement path *as a contiguous section of the placement range extending from a single earliest point to a single latest point.*

Following the decomposition we construct an *overlap graph* which captures all the overlapping relationships amongst the placement paths of all register pairs – the nodes in the graph represent individual placement paths and edges connect pairs of nodes that overlap with each other. This graph is then used to guide the placement decisions. A `clique` in the graph represents a group of placement paths then can be all covered by a single `SetMask` instruction. We iteratively select cliques from the graph and introduce `SetMask` instructions till all placement paths have been covered. The number of instructions introduced equals the number of cliques selected to cover the entire graph.

Next we illustrate the above steps of the algorithm using an example. In Figure 4.9, a *placement range* for `SetMask` R is shown. This placement range has three earliest points and one latest point. Upon decomposition, this placement range gives rise to three placement paths (P_0, P_1 and P_2). Now let us see how the *overlap graph* is constructed. Three nodes corresponding to the three paths are created and connected to each other as the three paths overlap. Let us consider presence of additional

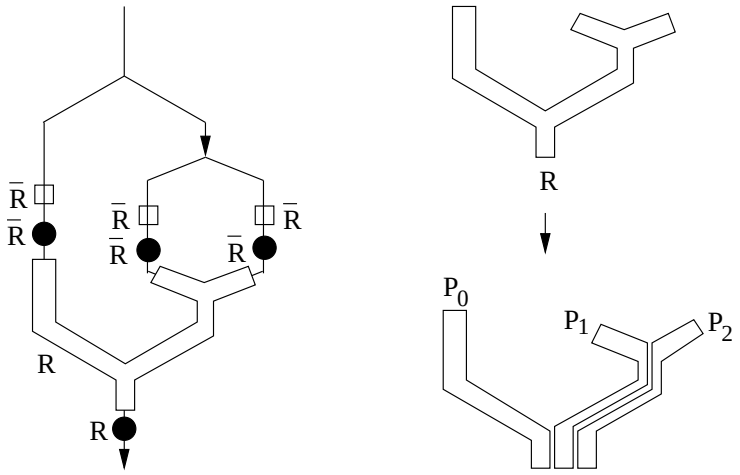


FIGURE 4.9. Splitting Placement Ranges into Placement Paths.

placement paths corresponding to registers $R1$, $R2$, and $R3$ giving rise to the overlap graphs shown in Figure 4.10. Now let see how the `SetMask` instructions are introduced. Cliques are used to cover the overlap graph and each clique corresponds to a `SetMask` instruction whose form is determined by the placement paths contained in the clique. This step is illustrated in Figure 4.11. The cliques chosen and the corresponding instructions introduced are shown. Note that in each case we have chosen the minimum number of cliques needed to cover the overlap graph. Minimum number of cliques correspond to minimum number of `SetMask` instructions.

Next we discuss how the cliques are selected. A greedy algorithm may be used that selects the *maximal clique* at each step. However, if there are multiple cliques of the same size that share nodes, then we need to decide which clique to pick as the choice will effect the total number of instructions introduced. For example, in the first overlap graph of Figure 4.12, there are two maximal cliques with three nodes – (P_1, P_2, R_2) and (P_0, P_1, P_2). If we select the first clique the graph is covered by two cliques while if we choose the second clique we need three cliques to cover the graph. We can further refine the greedy heuristic to choose between multiple maximal cliques. We can determine the minimum degree across nodes neighboring a maximal

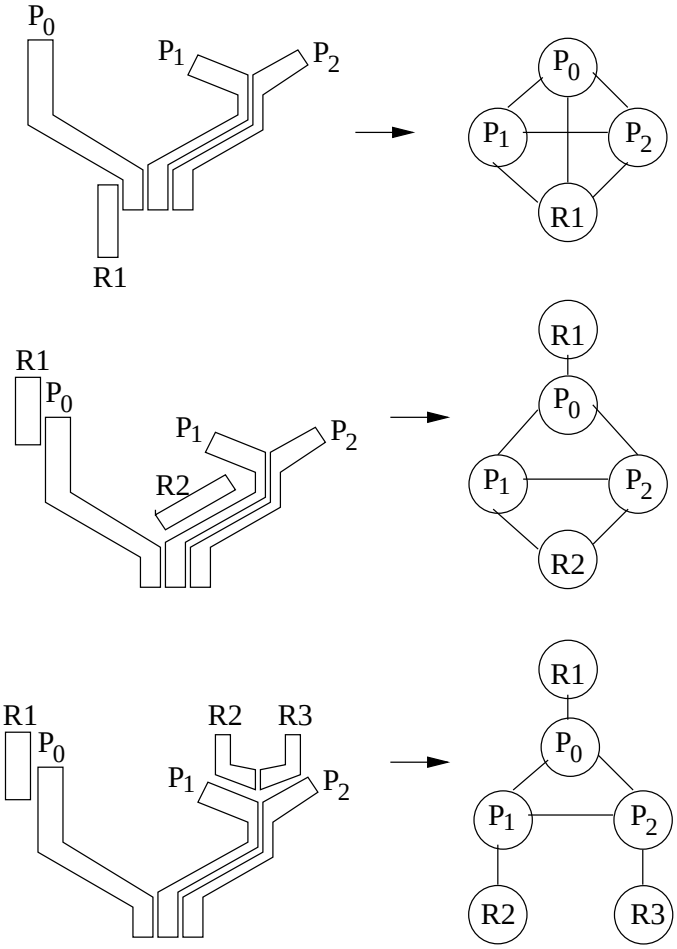


FIGURE 4.10. Overlap Graph.

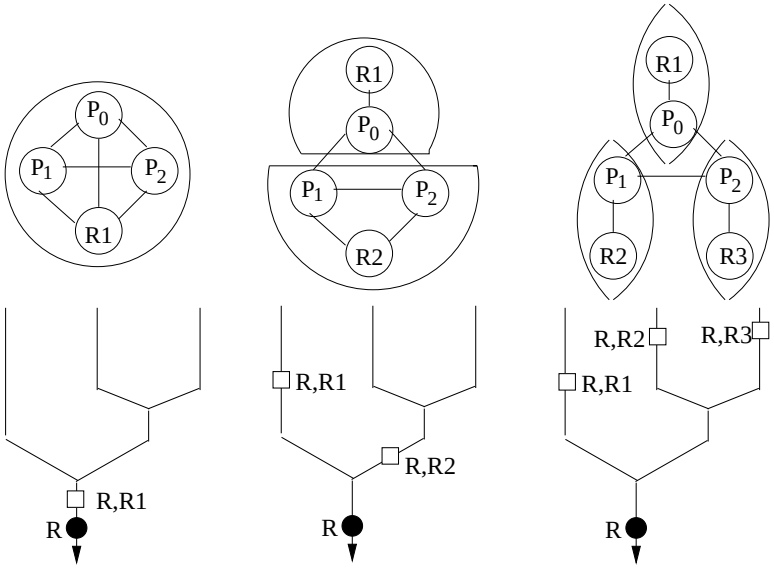


FIGURE 4.11. Clique Selection and Final Placement.

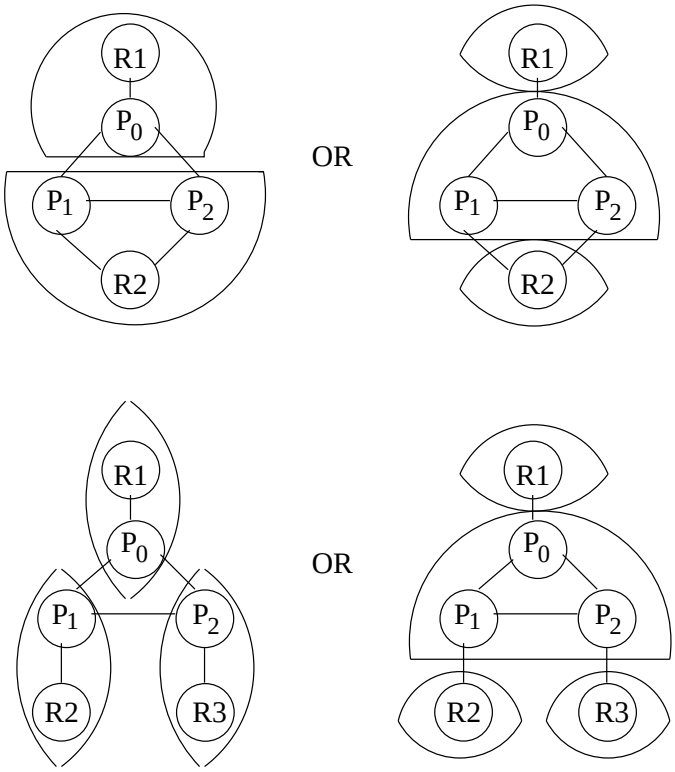


FIGURE 4.12. Clique Selection.

clique after the clique has been removed. The higher the degree the better it is because higher degree is likely to translate into larger future cliques being available. This approach will have the desired result of selecting (P_1, P_2, R_2) in the first example shown below. While we use the above approach, it is important to point out that this is a heuristic and thus it will not guarantee optimal results. Even if there is a unique maximal clique, always picking maximal clique does not necessarily result in fewest instructions. In the second example shown in Figure 4.12, if the do not choose the maximal clique (P_0, P_1, P_2) we can cover the graph using three cliques while if we choose the maximal clique (P_0, P_1, P_2) we need four cliques to cover the graph.

We would also like to mention that finding the maximal cliques is quite straightforward. We can examine each program point and count the number of placement paths to which that point belongs. This approach will identify all cliques and thus we can identify all maximal sized cliques. The cliques can then be prioritized by going back to the overlap graph and accessing the impact of selecting each of these cliques on degrees of neighboring nodes.

Final Placement Points

Decompose all *Placement Ranges* into *Placement Paths*.
Construct *Overlap Graph*:
  Nodes correspond to placement paths; and an Edge between a pair of nodes indicates that the corresponding paths overlap.
While *Overlap Graph* is not empty **do**
  Find *Maximal Cliques*.
  Select highest priority *Maximal Clique*.
  Remove selected clique from the *Overlap Graph*
    and insert *coalesced* `SetMask` instruction.
endwhile

FIGURE 4.13. Coalescing and Final Placement.

Finally we summarize the details of Step 3 in Figure 4.13. As we can see, first placement ranges are decomposed into placement paths. Next the overlap graph is constructed and then one by one cliques and selected and removed from the overlap

graph and corresponding `SetMask` instructions are inserted in the program.

4.3 Experiments

The goal of our experiments is two fold. First we would like to determine the benefit in performance that results from making use of `SetMask` instructions. Second we would like to determine the effectiveness of our presented algorithms in limiting the increase in code size. To carry out this experimentation we implemented the described techniques in our simulation and compilation environment. Then we ran the ARM, Thumb, and SetMask (Thumb code with `SetMask` instructions) versions of the programs and compared their performance and code size. We describe our implementation, experimental setup, followed by a discussion of the results.

Implementation The `xscale-elf gcc version 3.04` compiler used was built to create a version that supports generation of ARM and Thumb code. The above compiler already makes use of higher order registers as spill locations. In other words when no lower order registers are available, one is freed by spilling its contents into a higher order register that is free. Only if no registers are available values are spilled to memory. Therefore the consequence of limited access to higher order registers is generation of `MOV` instructions. We identify the points at which values are spilled into higher order registers and modify the code generated at these points so that the `MOV` instructions are eliminated. `SetMask` instructions are introduced instead. Then we apply the techniques described in this paper to minimize the `SetMask` instructions introduced. By making the register allocator aware of the extra registers made available, it is likely that spill code generated will be reduced. We are currently investigating this, the results shown here provide a lower bound on the performance improvement one can get using our approach. We evaluate the impact on performance by studying the effectiveness of our algorithms in eliminating `MOV` instructions as well

as total instruction and cycle counts.

Simulation Environment and Benchmarks A modified version of the Simplescalar-ARM [2] *simulator*, was used for experiments. It simulates the five stage Intel's SA-1 StrongARM pipeline [14]. The I-Cache configuration for this processor are: 16Kb cache size, 32b line size, and 32-way associativity, and miss penalty of 64 cycles (a miss requires going off-chip). The simulator was extended to support both 16-bit and 32-bit modes, the Thumb instruction set and the system call conventions followed in the `newlib` c library. This is a lightweight C library used on embedded platforms that does not provide explicit network, I/O and other functionality typically found in libraries such as `glibc`.

The *benchmarks* used are taken from the `Mediabench` [21] and `Commbench` [36] suites as they are representative of a class of applications important for the embedded domain. The benchmark programs used do not require functionality not present in `newlib`. A brief description of the benchmarks is given in Table 3.1. Code size being a critical constraint, all programs were compiled at -O2 level of optimization, since at higher levels code size increasing optimizations such as function inlining and loop unrolling are enabled.

Increase in Code Size Code is a critical constraint and we show here how our algorithms result in extremely small increases, if at all any, in code size. Figure 4.14 shows the code size for ARM, Thumb along with the code size by using `SetMask` instructions in the naive way described earlier (*Naive* `SetMask`) and after applying our optimization algorithms (`SetMask`). The increase in code size seen in the naive case has been cut back to levels almost equal to that of Thumb code. Thus, the `SetMask` instructions have a negligible cost in terms of code size increase.

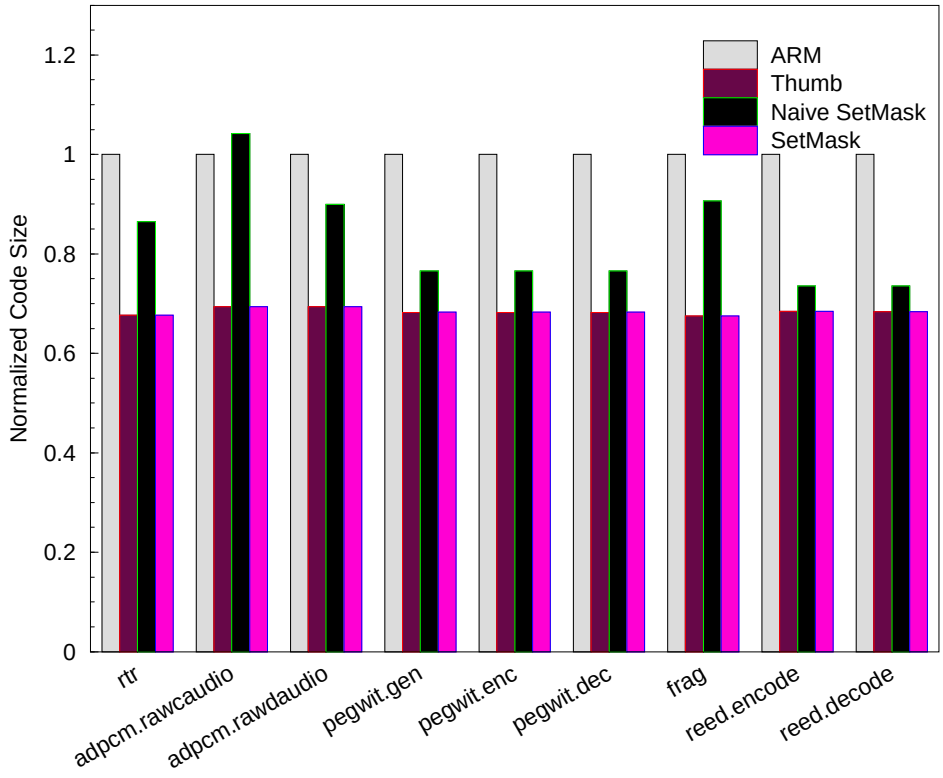


FIGURE 4.14. Normalized Code Size

Elimination of MOV Instructions By using the `SetMask` instruction we effectively cut down the number of MOV instructions executed at runtime. Recall that while `MOV` instructions have a single cycle execution cost, the `SetMask` instruction is coalesced with the preceding Thumb instruction using the Dynamic Coalescing Framework, hence having an execution cost of zero cycles. We measured the percentage of executed `MOV` instructions eliminated by making use of our techniques. The results are given in Table 4.1. As we can see, a significant percentage of MOV's (11.7%) introduced by the `gcc` compiler are eliminated by using `SetMask` instructions.

TABLE 4.1. Percentage of Executed MOVs Eliminated.

Program	MOVs Eliminated
rtr	21.1%
adpcm.rawcaudio	26.8%
adpcm.rawdaudio	0%
pegwit.gen	6.5%
pegwit.enc	27.6%
pegwit.dec	4.9%
frag	2.2%
reed.encode	10.1%
reed.decode	6.2%
Average	11.7%

We also measured the impact of eliminating `MOV`s on total instruction and cycle counts for the programs. Figure 4.15 shows the dynamic instruction count for ARM, Thumb and SetMask code. We achieve reduction of 0-19% in dynamic instruction count compared to Thumb code. `rtr` gives the the best result of improvement of 19% with other benchmarks giving moderate improvements and `adpcm.rawdaudio` giving no improvement over Thumb code. Figure 4.16 gives the cycle counts for ARM, Thumb and SetMask code. We achieve between 0-20% speedup in execution time in comparison to Thumb code. In some cases Thumb code is very close, sometimes faster, than the ARM code. This is due to the good cache behavior of Thumb code.

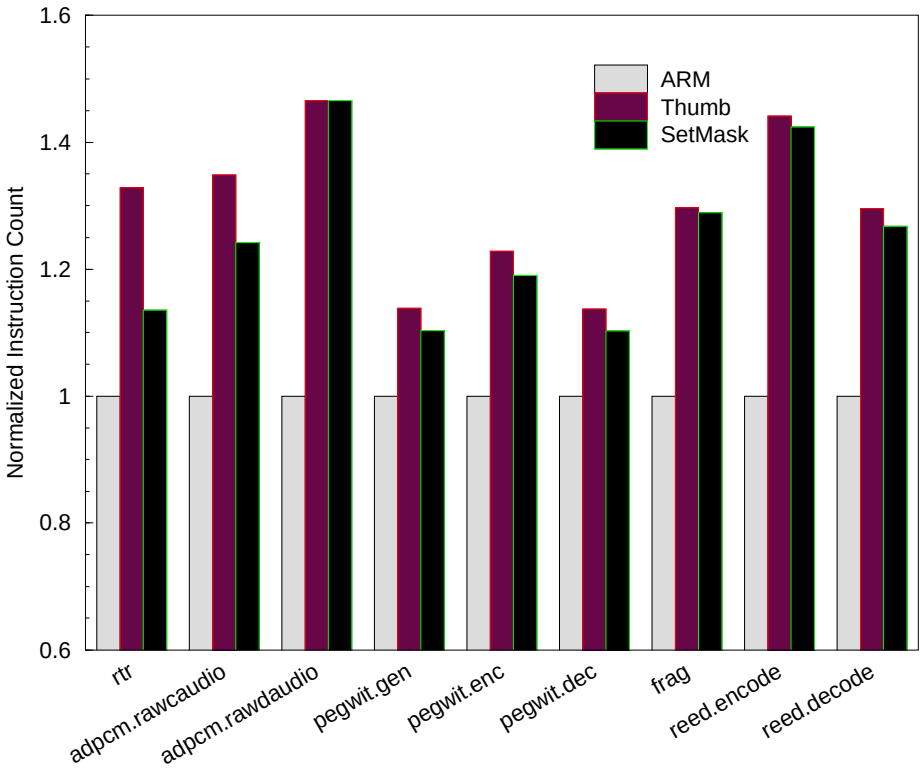


FIGURE 4.15. Normalized Instruction Counts.

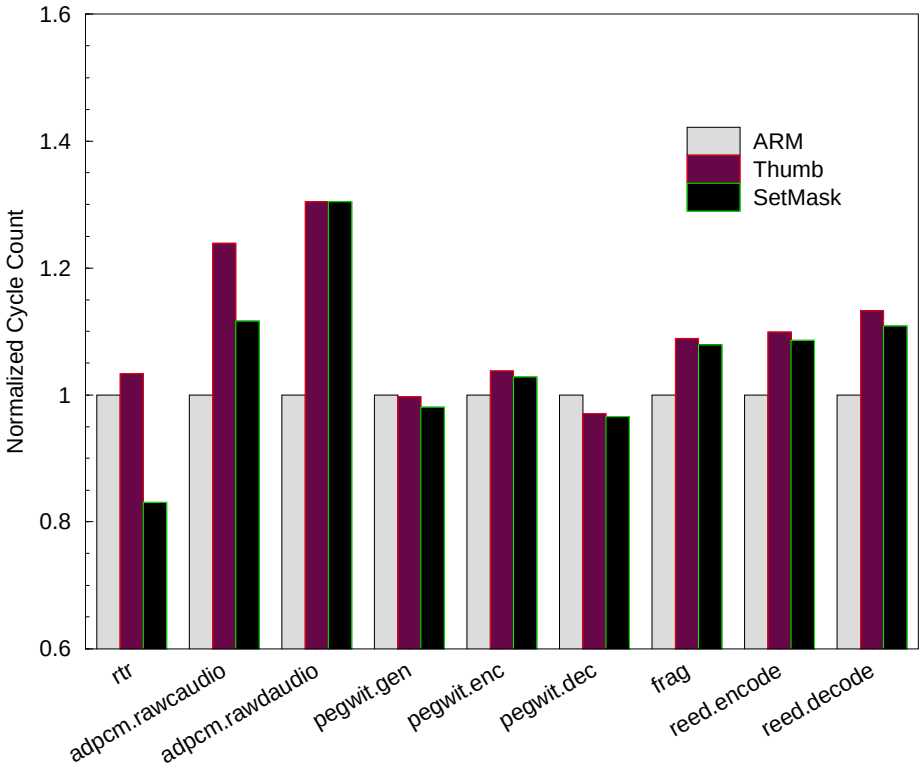


FIGURE 4.16. Normalized Cycle Counts.

`rtr` is a case where although the Thumb code is not faster than the ARM code, the SetMask code is much faster. Overall, the execution characteristics of SetMask are better than Thumb and comparable to ARM.

In summary, we have shown how with the effective use of `SetMask` instructions one can maintain the code size offered by Thumb code and achieve performance improvements at the same time.

4.4 Discussion

We have shown our approach on the ARM/Thumb platform with 16/8 registers respectively. Does this approach work for a larger register file? Our algorithms are applied post register allocation and register assignment. How does register assignment affect the `SetMask` instruction insertion? We address these questions here.

Scalability The SetMask mechanism relies on a bitmask which is indexed during the register file access and a mechanism to set this bitmask. When there is a notion of pairs of registers, like in our case, we can use a bitmask to activate different subsets of registers by toggling one bit for each pair. We had 8 addressable registers and 8 corresponding high registers and 8 bits of state corresponding to the 8 pairs. When we scale to a larger register file, we end up with many more non-addressable registers. In this case, we can no longer use one bit of state. We employ multiple bits of state. For instance, if we scaled to 32 registers with 8 addressable registers, we now have sets of 4 registers rather than pairs. Hence we use 2 bits of state rather than 1. We would need 2 `SetMask` instructions to set the 2 bitmasks. Hence the approach scales as long as we have 16-bit instruction encodings for the `SetMask` instructions.

Register Assignment The need for `SetMask` instructions arises when a register which is not in the current active subset is used. A different register assignment clearly changes the placement points for `SetMask` instructions. Hence one could decrease the

number of `SetMask` instructions introduced by changing the register assignment to minimize the number of switches between pairs of registers. This could precede our algorithms to minimize the number of `SetMask` instructions. However, as we have seen from our experiments, even without this preceding phase, our algorithms are able to keep the increased code size to a negligible amount. Hence while a different register assignment could precede our algorithms, we didn't find the need for it in our experiments.

4.5 Related Work

Prior work has studied the use of extra registers for high performance processors in various contexts. A 2-level hierarchical register file has been proposed in [38]. where they provide a small first level register file and larger second level register file enabling a larger register file with a larger number of ports. The first level register file has lower access latency compared to the second level register file allowing software pipelined loops to be executed more efficiently. Register Connection [18] has been proposed for superscalars to make more registers accessible to the compiler. A level of indirection is used to connect logical registers to the physical registers. Special register connect instructions are provided that can make changes to this mapping. The ILP available on superscalars allows the performance cost of the register connect instructions to be small. The additional code size introduced by these instructions is significant. While this it not much of a constraint on high performance machines, it is an important concern for embedded processors. Compiler Controlled Memory (CCM) [3] is another technique which tries to reduce the register pressure on the register file. It does so by incorporating a small memory close to the register file. The contents of this memory are managed by the compiler and used to handle spill code. Register windows have been used in the Tensilica Xtensa [35] and SPARC [34] architectures to avoid the saving and restoring of context during procedure calls. In all of the above techniques,

the size of the extended register file prohibits their use in embedded processors where power and cost are one of the main constraints.

Recently there have been proposals for the use of extra registers in embedded processors WIMS[28] proposes having several register windows and provides window management instructions which the compiler can use to swap register windows. During register allocation the virtual registers are partitioned into windows using a graph based partitioning algorithm. Code size increase due to window management instructions has not been considered in [28]. Our approach is also more flexible than register windows because it allows various subsets of 8 registers to be active. Differential Register Allocation[39] proposes encoding the register specifier using the difference between consecutive register accesses, allowing the compiler to allocate more registers than can be specified using a regular encoding. While this scheme has the same goals as our mechanism, it has the drawback of not being backward compatible with existing binaries because register specifiers cannot be correctly decoded without introducing delays. In [40] a small extended register file is used which is allocated at runtime. The aim is to reduce spill cost by dynamically choosing the extended register file over memory using compiler generated priorities. Offset fields in memory instructions are used to communicate these priorities to the hardware. This approach cannot be used to access existing high registers in Thumb state like our approach.

Prior work has also studied ISA design to allow access to higher registers. [20] proposes shrinking the destination register field of certain instructions and using this extra encoding space for other fields, partitioning the register file based on instruction type. They also describe a register allocation scheme for such an ISA. Mixed width ISAs can be exploited to allow access to both high and low registers by generating binaries with instructions from both the 32-bit and 16-bit instruction sets[19][11]. There have been several extensions to the ARM architecture[26][1] that seek to improve performance by allowing access the higher registers. Thumb-2[26] provides new 16-bit and 32-bit instructions in Thumb state. NEON[1] is a SIMD extension to

the ARM architecture that allows access to a special registers file for SIMD instructions. While our goal in this paper was to attack the global inefficiency of Thumb code, the SetMask mechanism can be implemented along with these proposals as it is orthogonal to these techniques.

4.6 Summary

In this chapter we attacked the problem of exposing the entire register file to the compiler and presented techniques to efficiently use it. We used the DIC framework to introduce an new AX instruction, *setmask*, that exposes the entire register file to the compiler. The supporting backward compatible register file design was also described. We then described the compiler algorithms that inserted *setmask* instructions to effectively use extra registers with insignificant increase in code size. Our experiments used the existing register allocator and showed how one can remove excessive *mov* instructions effectively. While the previous chapter used the DIC and AX framework to overcome shortcoming of a local nature, in this chapter we have used it to address a shortcoming that affects Thumb code globally.

Chapter 5
Dynamic Eager Execution

The techniques described in this chapter are different from the previous techniques in that they do not overcome inefficiencies of 16-bit code but rather exploit characteristics of dual width ISA architectures to provide features in 16-bit execution that are not possible in 32-bit execution. The extra fetch bandwidth is exploited to provide *Dynamic Eager Execution*. Dynamic Eager Execution consists of i) *Dynamic Delayed Branching* which improves the branch execution by dynamically creating a branch delay slot and scheduling an instruction in that slot and ii) *Dynamic 2-Wide Execution* which dynamically changes the issue width of the processor to issue upto 2 instructions simultaneously.

5.1 Delayed Branching and n-Wide Execution

In this section we will look at two techniques that have been used to speed up program execution, namely, *delayed branching* and *n-wide execution*. The first improves program performance by reducing the branch penalty while the second improves performance by maximizing the number of simultaneous instructions issued for execution thereby achieving superscalar execution. We will see what effects these two techniques have on performance and code size.

5.1.1 Minimizing Branch Penalty

Branch penalty is the penalty incurred by programs when the instructions fetched after the branch turn out to be from the wrong execution path. This penalty of having to flush all these instructions from the pipeline and and continue program

execution at the correct program address affects program execution negatively in two ways. Firstly, flushing the pipeline introduces pipeline bubbles into the program essentially stalling execution for those cycles. Secondly, the instructions fetched from the wrong path have unnecessarily used resources wasting energy.

Minimizing branch penalty is a well studied problem and has resulted in many solutions in many different contexts. Branch prediction is a commonly used technique, variants of which are implemented in most high performance processors. The idea is to predict the target of a branch and use the prediction to fetch instructions following the branch. An early study of branch prediction strategies was done in [33] followed by several recent studies including hybrid branch predictor designs [7] and branch predictor designs for recent high performance architectures [17]. While such complicated prediction schemes are not required for scalar pipelined architectures such as the ARM, the cost, space and energy budget of embedded processors usually precludes the use predictors.

An early solution to minimizing branch penalty was the idea of delayed branching. The architecture that supports delayed branching associates a fixed number of instruction slots after the branch instruction, usually one slot, to hold instructions from the program path preceding the branch. In other words the compiler is responsible for scheduling instructions that typically precede the branch into these slots. To ensure correct program execution the branch cannot have a dependence, whether it is a true or false data dependence or a control dependence, with instructions scheduled in the branch delay slots. The compilers inability to find such instructions will force it to schedule *nops* in the branch delay slots. We will examine this approach in detail since it does not have the limitations of branch prediction and can be implemented in embedded processors.

Figure 5.1 shows how delayed branching can improve program performance. Part (a) shows the timing diagram for a conventional pipelined architecture that does not support delayed branching and has a branch penalty of one cycle. Part (b) shows

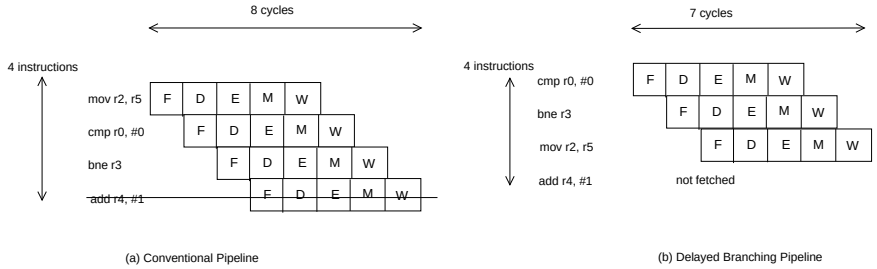


FIGURE 5.1. Delayed Branching Best Case

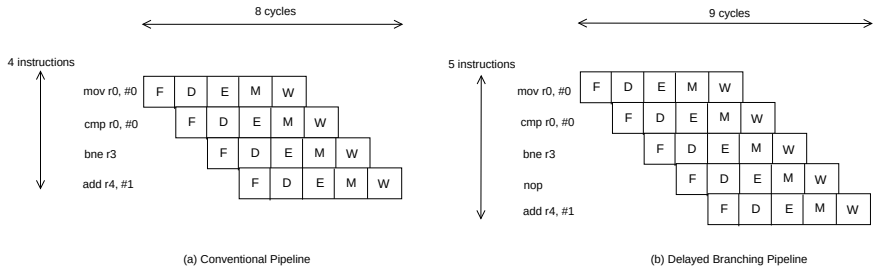


FIGURE 5.2. Delayed Branching Worst Case

the timing diagram for a delayed branching architecture with one branch delay slot. In addition to the number of execution cycles required for both architectures, the number of instructions generated by the compiler in both scenarios is shown.

In this case, lets assume the branch is taken. Hence the *add* instruction following the branch is fetched in the conventional pipeline and is flushed when the branch outcome is known. In the case of the delayed branching architecture, the independent *mov* instruction before the branch is scheduled in the branch delay slot and the *add* instruction is never fetched. The code generated for both cases is 4 instructions long not giving either architecture an advantage in terms of code size. When comparing performance however, we see that the delayed branching architecture executes one cycle earlier. In addition, from an energy standpoint, the delayed branching architecture does not wastefully execute any instructions.

Figure 5.2 shows how delayed branching can degenerate program behavior. Again, part(a) corresponds to the conventional architecture and part(b) corresponds to the

delayed branching architecture.

In this case, lets assume that the branch is not taken. Hence the *add* instruction is not wastefully fetched in the conventional architecture. In the case of the delayed branching architecture, the compiler cannot schedule any instruction in the delay slot (the branch now depends on the *move* instruction). So compiler will generate a *nop* which now takes up an extra cycle hurting performance and energy. In addition, the fact that the compiler has to generate the *nop* increases the number of instructions generated by 1. Hence the conventional architecture has the advantage over delayed branching for all three metrics in this case.

We have seen how delayed branching does not effectively address the branch penalty problem. At the same time techniques such as branch prediction are not applicable to embedded processors. Dynamic Delayed Branching overcomes these limitations as will be shown later.

5.1.2 Maximizing Instruction Issue and Execution

The performance of scalar pipelined machines can be improved by issuing and executing multiple instructions in parallel. Multiple issue processors can be classified into two categories: VLIW and Superscalar. The fundamental difference between the two is whether instructions for simultaneous issue are determined by the compiler statically or the hardware dynamically. VLIW processors are statically scheduled while superscalars schedule instructions dynamically.

VLIW processors rely on the compilers ability to exploit dependence information to statically schedule instructions such that multiple independent instructions can be scheduled in parallel. Such architectures have wide instruction words which can pack multiple instructions. The width determines the number of instructions that can be issued in parallel. These wide word formats also impose constraints on the types of instructions that can be issued in parallel to take resource contention into

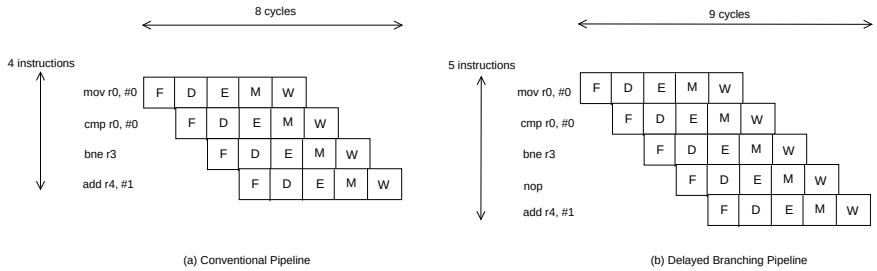


FIGURE 5.3. In-order Superscalars vs VLIW Processors

account. Since the burden of finding independent instructions falls on the compiler, the resulting architectures are not complex and make them suitable for embedded architectures.

Superscalar processors can be further classified into two kinds: in-order issue and out-of-order issue. In order issue processors issue instructions in the same order in which they arrive at the decode stage. In other words an instruction i that is statically scheduled by the compiler to execute later than an instruction j will never execute before j. The hardware merely tries to issue multiple consecutive independent instructions in parallel. Out-of-order issue processors on the other hand dynamically schedule instructions at runtime. Hence an instruction i that occurs later statically with reference to instruction j can be issued before j as long as there is not dependence. By eliminating false dependencies at runtime and being able to issue statically later instructions earlier, out-of-order issue processors can exploit significantly more ILP and hence improve performance significantly. However, this comes at the price of added complexity in hardware. Instruction Issue Windows have to be large enough to find independent instructions; the dependency checking logic in the issue stage is very complex; and to maintaining in-order commit to precise exceptions adds more hardware and complexity. This additional hardware cannot be justified in the embedded context and hence this avenue of exploiting ILP is not available.

We will now compare in-order superscalars with VLIW through the example shown in Figure 5.3. Both processors can issue upto 2 instructions each cycle. The sequence

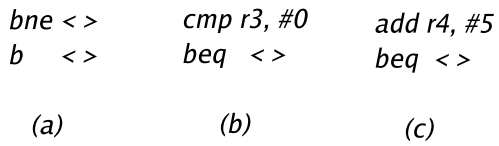


FIGURE 5.4. Cases for Dynamic Delayed Branching

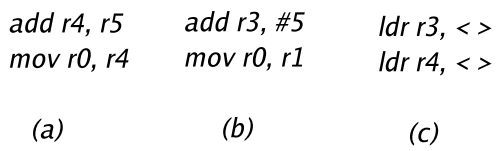


FIGURE 5.5. Cases for Dynamic 2-Wide Execution

of code is a chain of 4 dependent instructions. Hence they cannot be executed in parallel. The compiler for the VLIW processor has to schedule *nops* in each of the 4 wide words. While both processors take the same number of cycles, the VLIW schedule is twice as long as the superscalar schedule. Moreover, extra energy is spent fetching and executing *nops*. So with a little added hardware complexity in-order superscalars can exploit ILP without affecting code size.

Given that we have the extra fetch bandwidth and hence an instruction window of 2 instructions, we can perform in-order superscalar execution in Thumb. We will see in the next section how delayed branching and 2-wide execution can be performed in our dynamic eager execution microarchitecture.

5.2 Dynamic Eager Execution Microarchitecure

The Dynamic Eager Execution microarchitecture (DEE) eagerly executes branches early through dynamic delayed branching and eagerly executes 2 instructions in parallel when possible through dynamic 2-wide execution. In this section we we look at each of the components of DEE and how they perform eager execution.

First lets look at what functions need to be performed for dynamic eager execution. We will look at dynamic delayed branching and 2-wide execution separately.

Figure 5.4 illustrates the three cases that one has to deal with for dynamic delayed branching. All three code samples show 2 instructions which may be in the instruction buffer that need to be examined for delayed branching. The first example shows two branch instructions. Since only one is taken this sequence cannot use delayed branching. In the second case, the *cmp* instructions sets the zero flag which is used by the subsequent branch instruction. We cannot issue the branch early because of this dependence. The third case is the case when we can issue the branch early. This is because the there is no dependence with the previous instruction. Now consider Figure 5.5. This is an illustration of the cases when dynamic 2-wide execution can be performed. In the first case there is a dependence between the two instructions, hence they cannot be executed in parallel. The second case can be issued and executed in parallel since there is no dependence. The third case is interesting because even though there is no dependence between the two instructions they cannot be issued in parallel. This is because the memory bandwidth is fixed at 32-bits and only one memory operation can be serviced at a time. Notice that only 2 statically adjacent instructions can be executed eagerly. In other words, for instructions which may have more than one dynamic predecessor, it is sufficient to maintaining information only along the follow through path or the statically adjacent instructions.

We can now look, in detail, at the functions that need to be performed to carry dynamic eager execution. First we need to check if one of the instructions is a branch instruction. This determines whether we can perform dynamic 2-wide or dynamic delayed branching. Next we need to make sure the two instructions are independent. Additionally, in the case of delayed branching we need to ensure that only one of the instructions is a branch. In the case of 2-wide execution we need to ensure that atmost one instruction is a memory operation. Now we describe the Dynamic Eager Execution Microarchitecture in detail describing each part and the functions it accomplishes.

The independence table stores the independence information for instructions. In-

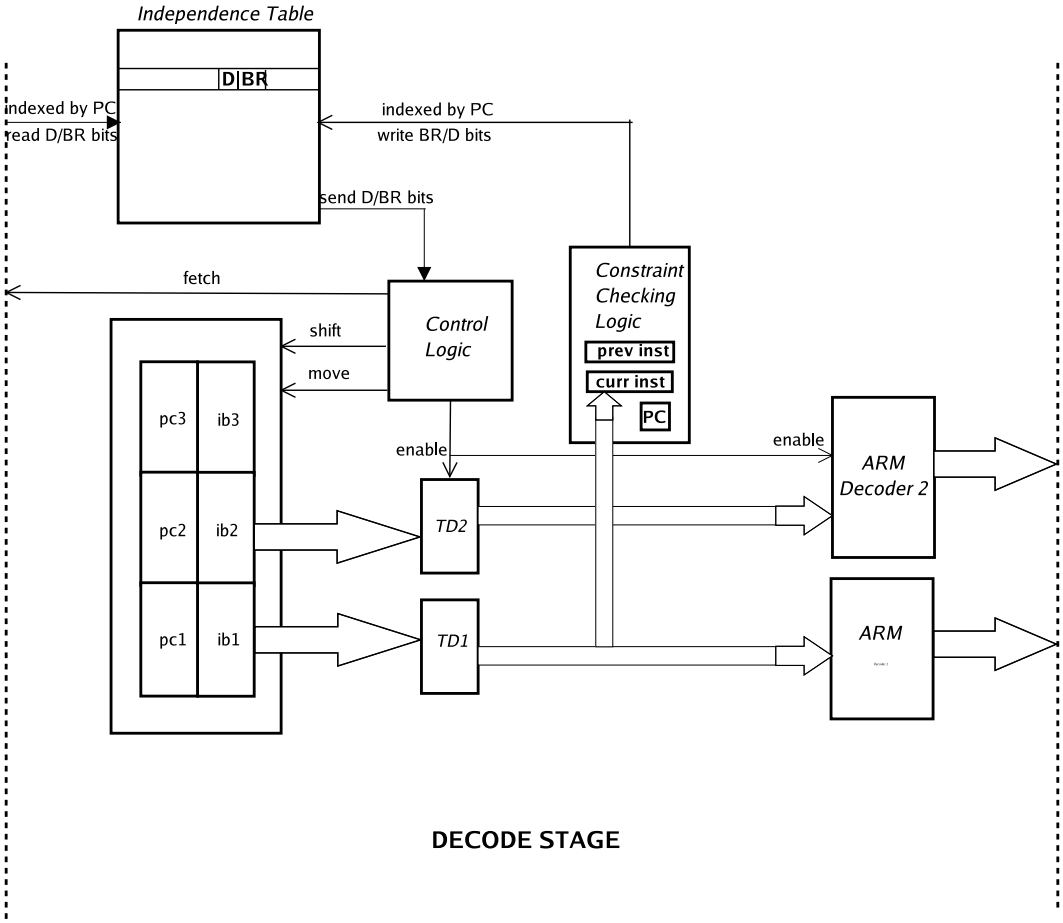
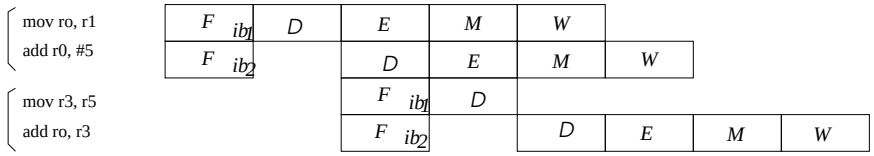


FIGURE 5.6. Dynamic Eager Execution Microarchitecture

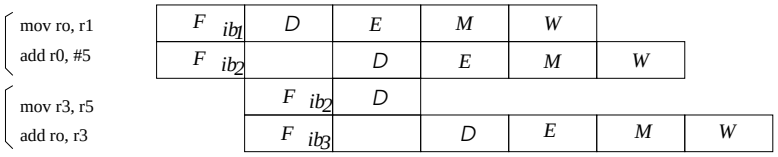
dependence information consists of dependence information between the instruction with the statically previous instruction and whether the instruction is a branch or not. DEE requires that independence information be known early enough so independent instructions can be decoded and issued eagerly. Performing a dependence check during decode would fall in the critical path of the processor which would increase cycle time. We avoid this by checking for independence before the instruction is decoded and while it is sitting in the instruction buffer. We do this by using the independence information gathered from a previous instance of this instruction which is stored in the independence table. The independence table is indexed by PC. Each instruction has two bits associated with it in the independence table. The *BR* bit indicates whether or not the instruction is a branch. The *D* bit indicates if there is a dependence between this instruction and it static predecessor. The independence table is managed like a direct mapped cache.

The control logic controls a) the movement of instructions in the instruction buffer, b) enabling/disabling the secondary thumb and arm decoders and c) the fetch logic. The instructions in the instruction buffer can shift by two, shift by one or be *skipped* to place the instruction from *ib3* in *ib1*. A shift by two occurs when two independent instructions are issued, and the instruction in *ib3* moves to *ib1*. A skip occurs when a branch is scheduled early hence placing the branch from *ib3* in *ib1* making *ib2* the branch delay slot. A shift by one occurs in the normal case, when neither dynamic delayed branching nor dynamic 2-wide execution takes place. Fetch is appropriately triggered to make sure the instruction buffer has 2 instructions at all times. In addition, once the 2-wide execution is detected the secondary thumb and arm decoders are enabled.

The instruction buffer is made 3 instructions wide. The reasons for doing this are similar to those for the DIC microarchitecture introduced in chapter 2. We need to ensure that two instructions are always available in the decode stage while another is being decoded. This is so the independence check can be performed a cycle early mak-



(a) 32 bit Instruction Buffer.



(b) 48 bit Instruction Buffer.

FIGURE 5.7. Extending the instruction buffer to 48 bits

ing full use of the lookahead capability. A 2-long buffer will not allow this. Moreover, a 3-long buffer ensures that we do not miss opportunities for eager execution due to alignment of instructions. This is shown in Figure 5.7. Instructions 2 and 3 can issue simultaneously, however a 2-long buffer constrains the lookahead to instruction that are word aligned while the 3-long buffer can lookahead beyond such boundaries. The same argument applies for dynamic delayed branching. In addition the instruction buffer is now augmented with PC information for each instruction in the buffer. This is to allow the constraint logic to correctly update the independence information for instructions in the independence table.

The constraint checking logic is responsible for updating the independence table. The two most recent instructions that have executed are checked. First, the PC of both instructions is checked to see if they are statically adjacent. Next the instructions are checked to see more than one of them is a branch. The check for atmost one load instruction is also performed. The dependency check is also performed here. All of these checks are aided by information available from the decoders. Once the

constraints have been checked the independence table is accessed using the PC of the curr instruction and updated accordingly. Constraint checking needs to be done only for those instructions that did not get eagerly executed.

In addition to the structures shown in Figure 5.6, the architecture has additional ALU units to enable parallel 2-wide execution. The register file supports 2 more read ports and 1 more write port to enable reading and writing of operands/results for two instructions at a time. Given that we are executing instructions eagerly, we need to ensure that we maintain precise exceptions. For example in the case of Dynamic Delayed branching, it is possible that the instruction scheduled in the branch delay slot can cause an exception. In the case of 2-Wide execution it is possible that one instruction is a memory op that misses in the cache and hence takes longer to finish while the other instruction causes an exception earlier. In both cases we commit results in order and report exceptions in order to maintain precise exceptions.

5.3 Experiments

The goal of our experiments is evaluate the performance improvement achievable using Dynamic Delayed Branching and Dynamic Eager Execution. Being a purely architectural technique, code size is not affected. Hence our focus will be on measuring improvements in dynamic instruction count and cycle count. Our comparisons will be with ARM and Thumb equivalents as we have done in previous chapters.

Experimental setup As in previous chapters, a modified version of the Simplescalar-ARM [2] *simulator*, was used for experiments. It simulates the five stage Intel's SA-1 StrongARM pipeline [14] with an 8-entry instruction fetch queue. The I-Cache configuration for this processor are: 16Kb cache size, 32b line size, and 32-way associativity, and miss penalty of 64 cycles (a miss requires going off-chip). Both Dynamic Delayed Branching and Dynamic 2-Wide execution were implemented as described in the pre-

vious section. As before all benchmarks were compiled at -O2 to keep code size small. The *benchmarks* used are taken from the `Mediabench` [21], `Commbench` [36] and `NetBench` [10] suites as they are representative of a class of applications important for the embedded domain.

Results Figure 5.8 shows the cycle counts for traditional Thumb execution and in using the DEE Microaarchitecture both normalized against the cycle counts for traditional ARM execution. We can ignore both code size and instruction counts as they remain the same. We present the improvements to cycle counts due the Dynamic Delayed Branching (DDB), Dynamic 2-Wide Execution (2W) and both together (DDB+2W).

Performance improvements for DDB range from 0.8% in the case of *frag* to 9.1% in the case of *pegwit.gen*. DDB has little performance improvement (around 1%) in the case of *adpcm* and *frag*. In the case of adpcm, due to tight loops, the number of fall through branches is low, providing little opportunity to exploit delayed branching. In the case of *frag*, branches are preceded by dependent *cmp* instructions, not allowing them to be scheduled in the branch delay slot.

Performance improvements for 2W range from 0.7% in the case of *reed.decode* to 13.7% in the case of *adpcm.rawc*. Very small performance improvements (around 1%) were seen in the case of *reed.encode*, *reed.decode*, *frag* and *rtr*. The small improvements are due to reasons - the presence of statically consecutive memory operations and small basic blocks. Consecutive memory operations cannot be executed in parallel in the dynamic 2-wide execution framework. Small basic blocks translate to more branches being seen ne fewer instructions that can be scheduled in parallel.

Performance improvement for DDB+2W range from 1.6% in the case of *frag* to 14.7% in the case of *adpcm.rawc*. The performance improvement is not strictly additive because there are cases when we have to choose between one or the other. For example, for a 3 instruction sequence of independent instructions that end with

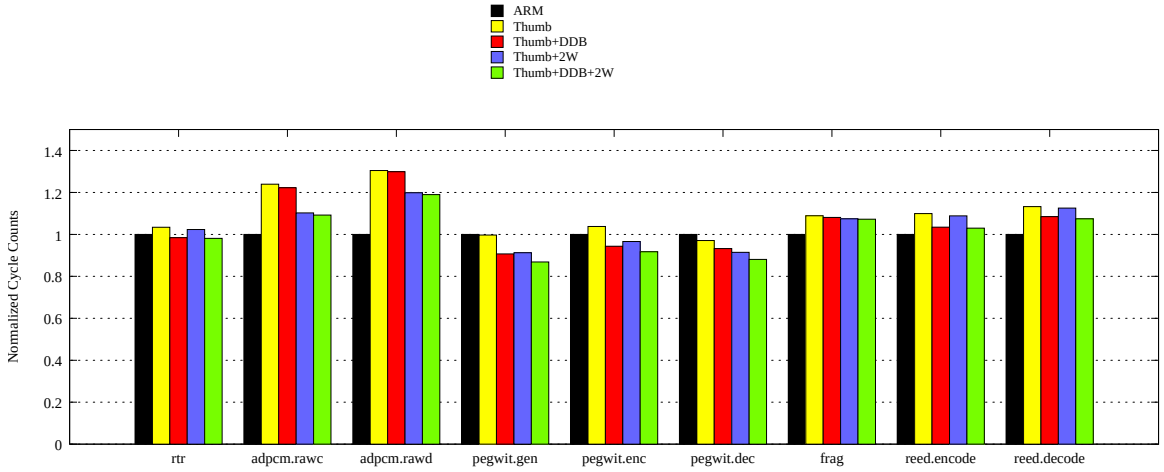


FIGURE 5.8. Cycle counts for traditional and DEE Thumb

a branch, we can perform 2W or DDB not both. *pegwit.gen* which gave individual performance improvements of 9.1% and 7.8% for DDB and 2W respectively gives a combined performance improvement of 13.1%.

5.4 Summary

In this chapter we explored a purely microarchitectural technique that exploits the extra fetch bandwidth of dual width ISA ARM processors to speed up execution. Dynamic Eager Execution - a framework that performs dynamic delayed branching and dynamic 2-wide execution was proposed. Dynamic Delayed Branching improves branch behavior, but does without the disadvantages of regular delayed branching. Dynamic 2-Wide Execution allows the processor to change the issue width dynamically allowing upto 2 instructions to be issued in parallel. The DEE framework is different from the DIC/AX framework in two ways. First, it is a purely architectural technique and does not require compiler support. Second, rather than trying to overcome the shortcomings of Thumb code, DEE seeks to provide performance improvement via techniques not viable for ARM code.

Chapter 6
Conclusion

In conclusion, let us revisit the main contributions of this dissertation and take a look at some future work.

6.1 Contributions

Dual Width ISA processors are a popular choice for the high performance embedded domain as they provide a choice between slow but small 16-bit code that can fit in small memories and fast 32-bit code that requires more memory. While this provides the programmer or compiler the flexibility to choose either small or fast code, it fails to provide the ideal case: small *and* fast code. To achieve this end, techniques were proposed to significantly improve the speed or performance of the small but slow 16-bit code without negatively affecting its small code size.

The underlying aspect of the techniques described in this dissertation is the more efficient utilization of existing resources in dual width ISA architectures for better execution of 16-bit code. In particular, two artifacts of dual width ISA designs, namely, extra fetch bandwidth and invisible registers, are exploited. The proposed Dynamic Instruction Coalescing Framework is an integrated compiler/microarchitecture platform aimed at improving the performance of Thumb code by overcoming the inefficiencies of the Thumb ISA. In addition, a purely microarchitectural technique, Dynamic Eager Execution, was proposed. *DEE* relies on the existing fetch bandwidth to provide eager execution not possible in 32-bit ARM state.

Dynamic Instruction Coalescing Framework The DIC/AX framework which provides the microarchitectural and ISA foundation to carry out the compiler opti-

mizations aimed at addressing the inefficiencies of Thumb code. The ISA was extended to accommodate Augmenting eXtensions or AX instructions. AX instructions allow the compiler to provide some augmenting information that is used to better execute the following instructions in the program. These instructions are executed by the Dynamic Coalescing architecture at zero cost by coalescing their execution with the following Thumb instruction. The AX instructions described were encoded using just one free opcode from the 16-bit instruction space. Several local optimizations and a form of predication that can be effected using appropriate use of AX instructions were described.

Local Optimizations with AX Various local optimizations served as the motivation for the design of the Dynamic Instruction Coalescing Framework. The compiler algorithms associated with these optimizations were described. The compiler algorithms were implemented in 3 phases after code generation. The first phase used AX to predicate branch hammocks effectively. The second phase sought out opportunities to replace pairs of thumb instructions with pairs of AX-Thumb instructions. These form the bulk of the AX instructions described and handle several specific peephole opportunities. The final phase replaced sequences of Thumb instructions in the function prologues and epilogues with a pair of AX-Thumb instructions improving performance by reducing the call overhead in Thumb programs. A comparison of the results with an approach proposed earlier, Profile Guided Mixed Code[19], showed that DIC/AX was more effective.

Global Optimization with AX A global optimization approach that made better use of the existing register file in dual width ISA processors was proposed. Using the DIC/AX framework, a new AX instruction *setmask* that exposes the entire register file to the compiler was introduced. *setmask* allows for more efficient use of registers. *setmask* introduces the notion of an active subset of registers. The cor-

responding changes to the register file design that implement the semantics of the *setmask* instruction were described. Efficient compiler algorithms to insert these *setmask* instructions to effectively use the newly exposed registers without increasing code size were also proposed.

Dynamic Eager Execution The *DEE Microarchitecture* described, goes beyond trying to address the shortcomings of the thumb ISA and try to improve execution by exploiting the extra fetch bandwidth available. By providing some microarchitectural enhancements and using the lookahead capability in Thumb state, a form of delayed branching and 2-wide execution was implemented. These techniques are more efficient than their traditional forms.

In summary, using the compiler and architectural techniques described here we can efficiently generate and execute 16-bit code, meeting the criteria of both code size and performance.

6.2 Future Work

This dissertation has focused on two primary metrics used to measure program execution: performance and code size. While these metrics will continue to be first class design constraints along with power and area, with shrinking feature sizes and new application domains, new metrics such as fault tolerance and security will become equally important for architectures and compilers.

New metrics result from changing demands of new application domains and/or changing physical properties of processors due to advances in semiconductor processing. The concept of an Augmenting eXtension serves as a platform to meet some of these demands. Dual-Width architectures are popular in the embedded domain making the techniques described in this dissertation more attractive when considering using AX to solve future problems. Specific program information such as encryption

keys for security or region of redundancy for fault tolerance can be specified using AX instructions. New microarchitectures/compiler methods can then use these AX instructions to implement faster encryption or better fault tolerance. While the AX instructions described in this dissertation have been done to fit into the the existing 16-bit ISA, designing a 16-bit instruction set with the AX in mind might yield considerably better instructions sets. Several AX instructions could share encodings if a hierarchy were built into the instructions. This would borrow from the setmask notion of having several active subsets to the instruction space where different subsets of AX instructions provide different functionality in a compiler determined fashion allowing fine-grain reconfigurability.

This dissertation introduces several new techniques that form a firm foundation for future work.

References

[1] ARM Inc. *ARM NEON Technical Data Sheet*, March 2004.

[2] D. Burger and T.M. Austin. The simplescalar toolset version 2.0. *Computer Architecture News*, pages 13–25, June 1997.

[3] Keith D. Cooper and Timothy J. Harvey. Compiler-controlled memory. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 2–11. ACM Press, 1998.

[4] Marc L. Corliss, E. Christopher Lewis, and Amir Roth. The implementation and evaluation of dynamic code decompression using dise. *Trans. on Embedded Computing Sys.*, 4(1):38–72, 2005.

[5] Saumya Debray and William Evans. Profile-guided code compression. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2002.

[6] Saumya K. Debray, William Evans, Robert Muth, and Bjorn De Sutter. Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems*, 22(2):378–415, 2000.

[7] Marius Evers, Po-Yung Chang, and Yale N. Patt. Using hybrid branch predictors to improve branch prediction accuracy in the presence of context switches. In *ISCA*, pages 3–11, 1996.

[8] Daniel H. Friendly, Sanjay J. Patel, and Yale N. Patt. Putting the fill unit to work: Dynamic optimizations for trace cache microprocessors. In *MICRO31*, Dec. 1998.

[9] S. Furber. *ARM System Architecture*. Addison-Wesley, 1996.

[10] W.H. Mangione-Smith G. Memik and Hu. Netbench: A benchmarking suite for network processors. In *IEEE International Conference on Computer-Aided Design*, pages 39–42. IEEE, November 2001.

[11] A. Halambi, A. Shrivastava, P. Biswas, N. Dutt, and A. Nicolau. An efficient compiler technique for code size reduction using reduced bit-width isas. In *DATE '02: Proceedings of the conference on Design, automation and test in Europe*, page 402, Washington, DC, USA, 2002. IEEE Computer Society.

[12] Shiliang Hu and James E. Smith. Using dynamic binary translation to fuse dependent instructions. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 213, Washington, DC, USA, 2004. IEEE Computer Society.

[13] Intel Corporation. *The Intel XScale Microarchitecture Technical Summary*, 2000. ftp://download.intel.com/design/intelxscale/XScaleDatasheet4.pdf.

[14] Intel Corporation. *SA-110 Microprocessor Technical Reference Manual*, 2000. ftp://download.intel.com/design/strong/applnots/27819401.pdf.

[15] Intel Corporation. *The Intel PXA250 Applications Processor - A White Paper*, February 2002.

[16] Quinn Jacobson and James E. Smith. Instruction pre-processing in trace processors. In *HPCA*, pages 125–129, 1999.

[17] Daniel A. Jimenez. Piecewise linear branch prediction. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 382–393, Washington, DC, USA, 2005. IEEE Computer Society.

[18] Tokuzo Kiyohara, Scott Mahlke, William Chen, Roger Bringmann, Richard Hank, Sadun Anik, and Wen-Mei Hwu. Register connection: a new approach to adding registers into instruction set architectures. In *Proceedings of the 20th annual international symposium on Computer architecture*, pages 247–256. ACM Press, 1993.

[19] A. Krishnaswamy and R. Gupta. Profile guided selection of arm and thumb instructions. In *Proceedings of the ACM SIGPLAN Joint Conference on Languages Compilers and Tools for Embedded Systems & Software and Compilers for Embedded Systems (LCTES/SCOPES)*, pages 55–64, Berlin, Germany, June 2002. ACM.

[20] Y-J. Kwon, X. Ma, and H.J. Lee. Pare:instructions set architecture for efficient code size reduction. In *Electronics Letters*, pages 2098–2099, 1999.

[21] C. Lee, M. Potkonjak, and W.H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 330–335, Research Triangle Park, North Carolina, December 1997.

[22] Sheayun Lee, Jaejin Lee, Sang Lyul Min, Jason Hiser, and Jack W. Davidson. Code generation for a dual instruction set processor based on selective code transformation. In *SCOPES*, pages 33–48, 2003.

[23] Charles Lefurgy, Peter Bird, I-Cheng Chen, and Trevor Mudge. Improving code density using compression techniques. In *IEEE/ACM Symposium on Microarchitecture (MICRO)*, December 1997.

[24] Haris Lekatsas and Wayne Wolf. Code compression for embedded systems. In *Design Automation Conference*, pages 516–521, 1998.

[25] H. McGhan and M. O'Connor. Picojava: A direct execution engine for java bytecode. *IEEE Computer*, pages 22–30, October 1998.

[26] R. Phelan. Improving arm code density and performance. June 2003.

[27] A. Qasem, D. Whalley, X. Yuan, and R. van Engelen. Using a swap instruction to coalesce loads and stores. In *Proceedings of the European Conference on Parallel Computing*, pages 235–240, August 2001.

[28] Rajiv A. Ravindran, Robert M. Senger, Eric D. Marsman, Ganesh S. Dasika, Matthew R. Guthaus, Scott A. Mahlke, and Richard B. Brown. Increasing the number of effective registers in a low-power processor using a windowed register file. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES-03)*, pages 125–136, 2003.

[29] R. Razdan and M. D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 172–80, 1994.

[30] G. Reinman and N. Jouppi. An integrated cache timing and power model. *Technical Report, Western Research Lab*, 1999.

[31] K. Clarke S. Segars and L. Goudge. Embedded control problems, thumb and the arm7tdmi. *IEEE Micro*, pages 22–30, October 1995.

[32] S. Segars. Low power design techniques for microprocessors. February 2001.

[33] James E. Smith. A study of branch prediction strategies. In *ISCA '81: Proceedings of the 8th annual symposium on Computer Architecture*, pages 135–148, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.

[34] SPARC International. *The SPARC architecture manual: Version 8*. Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1992.

[35] Tensilica Inc. *Xtensa Architecture and Performance*, September 2002.

[36] T. Wolf and M. Franklin. Commbench - a telecommunications benchmark for network processors. In *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 154–162, April 2000.

[37] Andrew Wolfe and Alex Chanin. Executing compressed programs on an embedded risc architecture. In *Proceedings of the 25th annual international symposium on Microarchitecture*, pages 81–91, Portland, Oregon, United States, 1992.

[38] Javier Zalamea, Josep Llosa, Eduard Ayguad, and Mateo Valero. Two-level hierarchical register file organization for vliw processors. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 137–146. ACM Press, 2000.

[39] Xiaotong Zhuang and Santosh Pande. Differential register allocation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 168–179, New York, NY, USA, 2005. ACM Press.

[40] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. Hardware-managed register allocation for embedded processors. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools*, pages 192–201. ACM Press, 2004.