Instruction Coalescing for 16-bit Code

Arvind Krishnaswamy Rajiv Gupta

Abstract

In the embedded domain, memory usage and energy consumption are critical constraints. Embedded processors such as the ARM and MIPS provide a 16-bit instruction set (called Thumb in the case of the ARM cpu family) in addition to the 32-bit instruction set to address these concerns. Using 16-bit instructions one can achieve code size reduction and I-cache energy savings at the cost of performance.

This paper presents a novel approach that enhances the performance of 16-bit Thumb code. We have observed that throughout Thumb code there exist Thumb instruction pairs that are equivalent to a single ARM instruction. We have developed enhancements to the processor microarchitecture and the Thumb instruction set to exploit this property. We enhance the Thumb instruction set by incorporating *Augmenting eXtensions* (AX). A Thumb instruction pair that can be combined into a single ARM instruction is replaced by an AXThumb instruction pair by the *compiler*. The AX instruction is coalesced with the immediately following Thumb instruction to generate a single ARM instruction at decode time. The enhanced microarchitecture ensures that coalescing does not introduce pipeline delays or increase cycle time thereby resulting in reduction of both instruction counts and cycle counts. Using AX instructions and coalescing hardware we are also able to support efficient predicated execution in 16 bit mode.

Keywords - embedded processor, 32-bit ARM ISA, 16-bit Thumb ISA, code size, energy, performance, AX instructions, instruction coalescing.

1 Introduction

More than 98% of all microprocessors are used in embedded products, the most popular among them being the ARM family of embedded processors [5]. The ARM processor core is used both as a macrocell in building application specific system chips and standard CPU chips.[2] (e.g., ARM810, StrongARM SA-110 [3], XScale [4]). In the embedded domain, in addition to having good performance, applications must execute under constraints of limited memory and low energy consumption. Dual instruction set processors, such as the ARM and MIPS, provide a unique opportunity for code size reduction by supporting a 16 bit instruction set along with the 32-bit instruction set. The 16-bit instruction provides a subset of the functionality provided by the 32-bit instruction set. Hence one can achieve good code size reduction using 16-bit code, but we pay a performance penalty since, for a given program, the number of 16-bit instructions needed is much more than the number of 32-bit instructions. In this paper we describe a technique, based on the ARM architecture, that reduces this performance gap between 16-bit and 32-bit code.

1.1 32-bit ARM Code vs 16-bit Thumb Code

Here we illustrate the tradeoffs present in the 32-bit ARM and 16-bit Thumb instruction sets to motivate our approach. The data in Figure 1 compares the ARM and Thumb codes along three metrics: Instruction Count, Code size and I-cache fetches. 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%. In prior work [6] it is shown that this substantial increase in the number of instructions executed by the Thumb code more than offsets the improved I-cache behavior of the Thumb code. Therefore the net result is higher cycle counts for the Thumb code in comparison to the ARM code. While we observe that by using Thumb code we nearly always save I-cache energy as a result of fewer fetches, the increase in instruction counts increases the energy consumed in other parts of the processor.

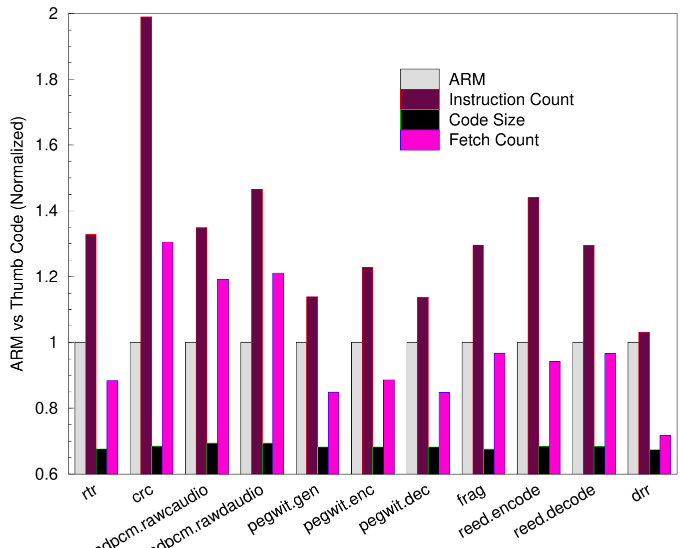


Figure 1: ARM vs Thumb Code

On further analysis we were able to determine that the dynamic instruction count increase is mainly due to increase in three categories of instructions: Branches, ALU operations, and MOVs. The reasons for increase in these categories are elaborated in our discussion of the AX instructions. In the above situations we are able to find short sequences of Thumb instructions that can be easily replaced by shorter sequences of ARM instructions. One could generate a mixed binary using both ARM and Thumb instructions, however, the overhead of explicit switching between 16-bit mode and 32-bit mode for short sequences negates the benefit of mixed code, as has been shown in [6].

1.2 Contributions

This paper presents a novel approach that enhances the Thumb instruction set to enable it to perform like ARM code. These enhancements allow patterns of Thumb instructions to be translated into ARM equivalents at runtime without requiring explicit switching of processor mode. We enhance the Thumb instruction set by incorporating *Augmenting eXtensions* (AX). Augmenting instructions are a new class of instructions which are entirely handled in the decode stage of the processor, not going through the remaining stages of the pipeline. Each AX instruction is coalesced with the following non-AX instruction in the program during the decode stage of the processor, where the translation of Thumb instructions into ARM instructions takes place. The *compiler* replaces patterns of Thumb instructions by equivalent sequences of AXThumb instructions. The *decode stage* is redesigned to detect augmenting instructions and perform coalescing to generate more efficient ARM instructions for execution. The distinctive characteristics of our approach are:

- *Coalescing Without Pipeline Delays.* When coalescing is performed, no additional pipeline bubbles are introduced as instruction fetching does not fall behind. When two instructions are coalesced during execution of AXThumb code, two additional Thumb instructions are available for decoding in the very next cycle.

- *Simple Coalescing Hardware.* By placing the responsibility of identifying instruction coalescing opportunities on the compiler, AX enables us to achieve coalescing using simple modifications to the decode stage. While a compiler can easily recognize coalescing opportunities, and appropriately mark them using AX instructions, the hardware cannot do so either easily or safely.

- *Supporting Predication in Thumb.* AX not only incorporates predicated execution into the Thumb instruction set, but simple support in the decode stage allows an implementation of predication which is even more efficient than the ARM implementation of predication.

- *Avoiding Mode Switching.* Our approach does not require explicit switching of processor modes since the fetched instructions are always 16 bit AXThumb instructions.

The remainder of the paper is organized as follows. Section 2 describes the concept of augmenting instructions and the coalescing mechanism for handling these instructions. We also show how this novel coalescing mechanism can with a minor modification allow us to incorporate a highly effective method for executing predicated code. We also provide details of the set of augmenting instructions we have developed. Section 3 presents the results of our evaluation. Conclusions are given in section 4.

2 Instruction Coalescing

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

To illustrate the key concepts of our approach we use a simple example. In the code above we show 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, as shown above two Thumb instructions are required to achieve the effect of one ARM instruction. We would like to coalesce the 2 16-bit instructions into 1 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. In our example the Thumb code uses a temporary register `rtmp`. If instruction coalescing is performed, `rtmp` is no longer needed and therefore its contents will not be changed. Therefore, 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.

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 the *Augmenting eXtensions* (AX). In the AXThumb code the first instruction is an augmenting instruction which is not executed but rather 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 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 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 the 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.

2.1 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 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.

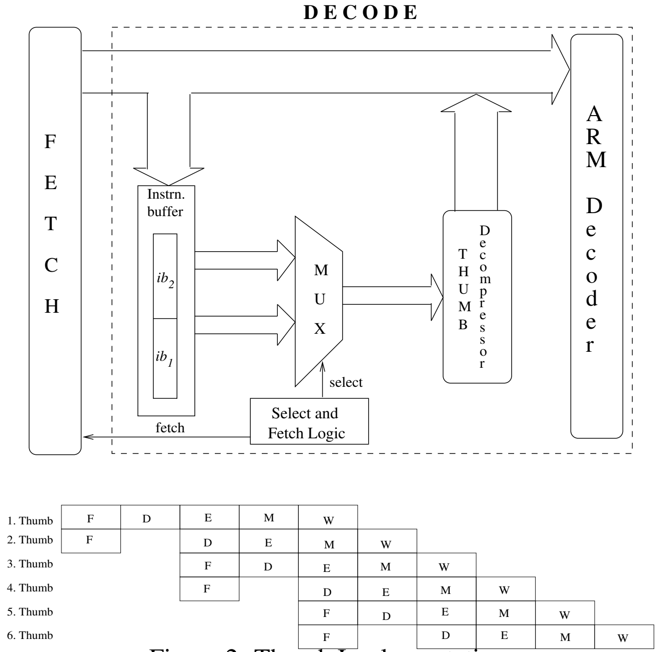


Figure 2: Thumb Implementation.

2.1.1 Instruction Coalescing

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, 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 mode a 32 bit instruction is directly fed to the ARM decoder. However, in the Thumb mode the 32 bits are held in an *instruction buffer* and the two Thumb instructions that it contains 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. Since every time a word is fetched we get two Thumb instructions, fetch needs to be carried out 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.

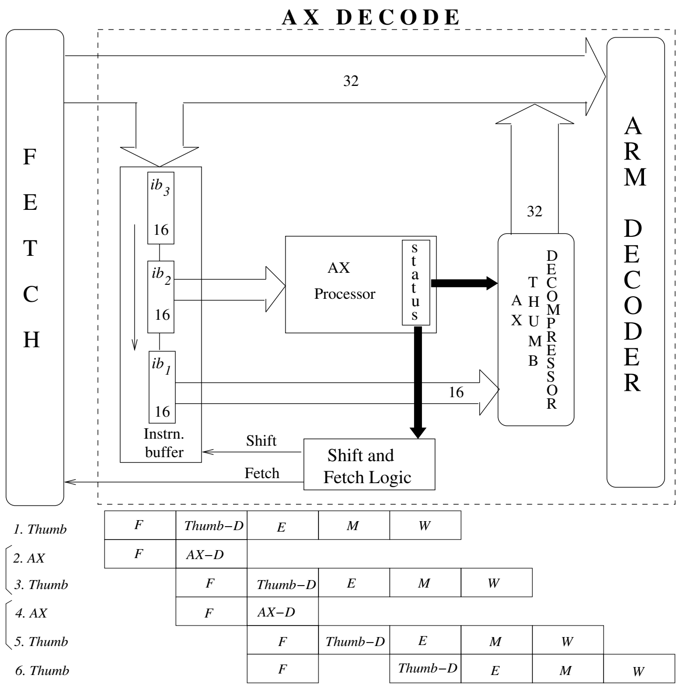


Figure 3: AXThumb Implementation.

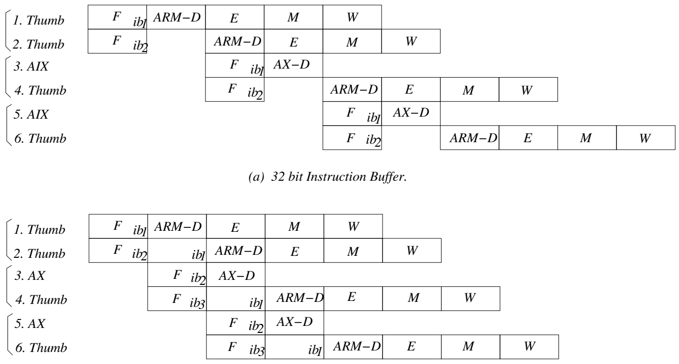
The overall operation of the hardware design shown in Figure 3 is as follows. The *instruction buffer* in the decode stage is modified to exploit the extra fetch bandwidth to keep at least two instructions in the buffer at all times. Two consecutive instructions, one Thumb instruction and a following 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 32-bit register. Hence during a thread switch it is sufficient to save the state of this status register along with other state to ensure correct execution when this thread resumes.

There are three important points to note about the above operation. First, as shown by the pipeline timing diagram in Figure 3, 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 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.

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 therefore 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).

Now we illustrate the need to expand the instruction buffer to hold up to three instructions. In Figure 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 4(b) when we expand the buffer to 48 bits, the instructions are deposited by the fetch sooner and thereby causing the AX instruction(s) and the preceding Thumb instruction(s) to be simultaneously present in the buffer. Therefore the AX instructions are now handled for free.

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



(a) 32 bit Instruction Buffer.

(b) 48 bit Instruction Buffer.

Figure 4: Delivering Instructions to Decode Ahead for Overlapped Execution.

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 absence of branches the above condition is satisfied because at the beginning of the decode cycle the buffer definitely contains two instructions and 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 must be 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 and therefore 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 the 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 AX instruction actually adds a cycle to the execution. In other words the benefit of introducing the AX instruction is lost. When an AXThumb pair replaces a Thumb pair, the second Thumb instruction in the AXThumb pair need not be the same as the

[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.

second Thumb instruction in the Thumb instruction pair. Hence one cannot allow an AX instruction in ib_1 by issuing a nop when an AIX 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.

2.1.2 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 AX-Thumb 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 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.

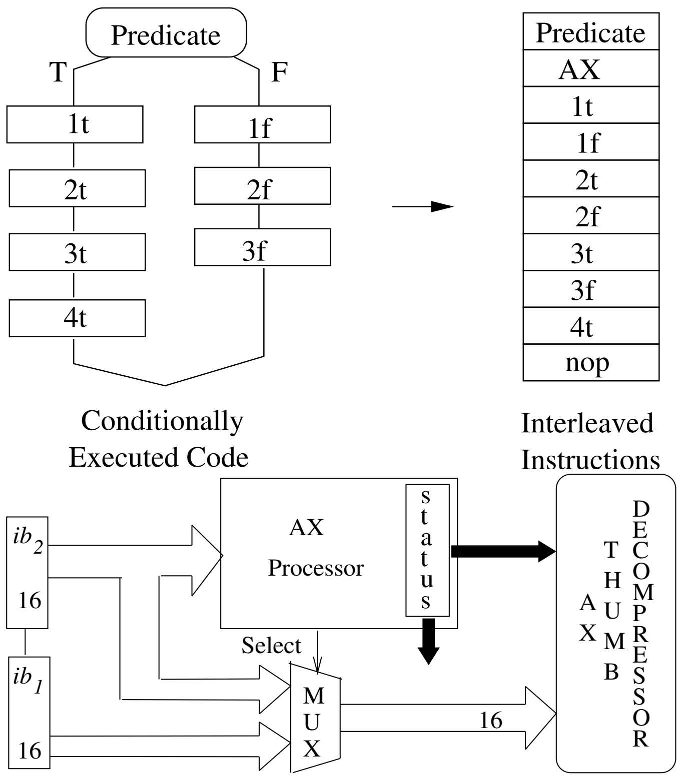


Figure 5: Predication in AXThumb.

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 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 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 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.

The above approach for executing predicated code is more effective than doing so in the ARM mode. In ARM mode 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 mode instructions arrive in the decode stage early while the same is not true for ARM.

2.2 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 first show how we use exactly one free instruction in the free opcode space of the Thumb instruction set to implement AX instructions. We describe these instructions next 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 effect 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.

2.2.1 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 below. 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]

2.2.2 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 kind 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 `r1` using the `mov` and `neg` instructions in the Thumb mode. The address computation of `rbase + r1` is also carried out by a separate instruction in the Thumb mode. Therefore one ARM instruction is replaced by 4 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 specified as part of `str` 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 rotated by the *rotate* amount to generate a 32 bit constant. In Thumb mode the immediate operand is only 8 bits and therefore the *rotate* amount cannot be specified. An additional ALU instruction is used to generate the large constant 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`

2.2.3 Predication - Branch Instructions

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 `rsbne` 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
rsbne r6, r6, r1
rsbne 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
sub r6, r6, r1
sub r5, r5, r2
```

OR

```
cmp r3, #0
add r6, r6, r1
add 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 2 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) are 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 in the `setpred` instruction.

This form of predication could also reduce the number 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
```

2.2.4 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 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	AXThumb
`ldr reg, [Hreg, #offset]`	`setsource Hreg` `ldr reg, [_, #offset]`
Corresponding Thumb	**Coalesced ARM**
`mov Lreg, Hreg` `ldr reg, [Lreg, #offset]`	`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 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 is shown below. It 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	AXThumb
`and reg1, reg1, #imm`	`setimm #imm`
Corresponding Thumb	`and reg1, _`
`mov rtmp, #imm`	Coalesced ARM
`and reg1, rtmp`	`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.

2.3 Compiler Support: AX Postpass

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. The algorithms for each of the three phases along with code examples are described in detail next.

2.3.1 Phase 1

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

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

In our example, when we examine the AXThumb code, we observe that the condition in this case is `eq` and `count` is 2 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) are 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.

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. However this form of predication can be applied only to small branch hammocks 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 Figure 2.3.1. The `CFGs` for the code example described above, before and after the transformation are shown in Figure 2.3.1.

2.3.2 Phase 2

The code segment shown below 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    : 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
```

Figure 6: SetPredicate

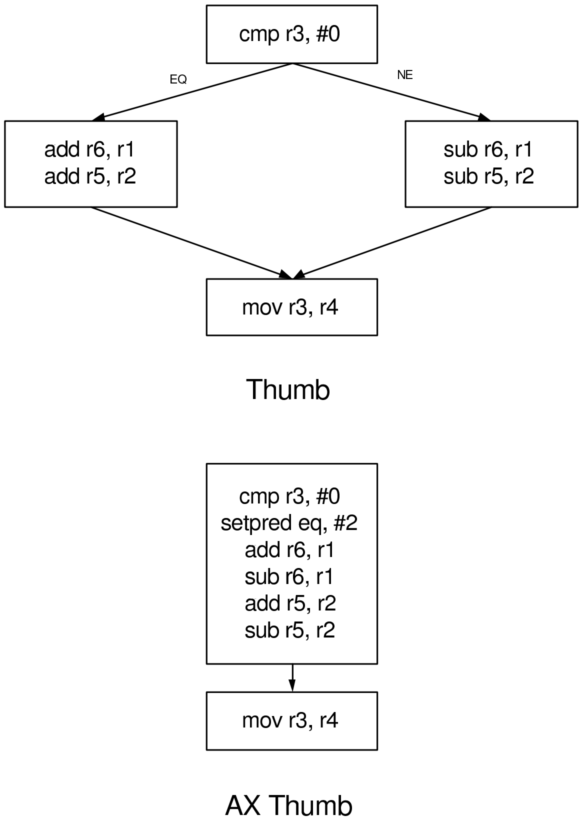


Figure 7: Predication

Since these transformations are local to a basic block, the algorithm shown in Figure 2.3.2 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 Bread-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 2 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 Figure 2.3.2 and the DAG for our example, before and after the transformation is shown in Figure 2.3.2.

2.3.3 Phase 3

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 below illustrates the need for a `setallhigh` instruction. Since only low registers can be accessed in Thumb mode, the saving and restoring of context at function 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.

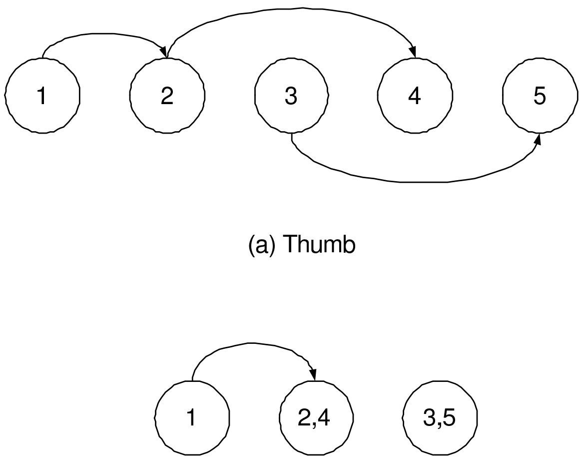
Thumb Code
(1) push [r4, r5, r6, r7]
(2) mov r4, r8
(3) mov r5, r9
(4) mov r6, r10
(5) mov r7, r11
(6) push [r4, r5, r6, r7]

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 alogrithm detects such sequences during a Breadth-First traversal of the DAG. The dependence in the DAG is between the push instructions and

```
input  : Basic Block DAG D with nodes numbered according to the topological
         order of the instruction schedule
output : Basic Block DAG D with Coalesced Nodes to indicate AXThumb instruc-
         tion 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
```

Figure 8: DAG Coalescing for generic AX instructions



(a) Thumb

(b) AX Thumb

Figure 9: Phase 2

the move instructions as shown in Figure 2.3.3. The move instructions are siblings with predecessor and successors as the push instructions in the DAG. This condition is checked for as shown in Figure 2.3.3. 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 succesor node which is the push/pop instruction to ensure that two instructions are adjacent to each other in the instruction schedule.

```
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
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
                if not movLoHi(m) | not PushOrPopListHi(Succ(m)) |
                numSuccs(m) ≠ 1 then
                    isReplacable = FALSE
                end
            end
            /* Remove mov's 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
```

Figure 10: DAG Coalescing for setallhigh AX instructions

3 Experimental Results

Experimental setup A modified version of the Simplescalar-ARM [1] *simulator*, was used for experiments. It simulates the five stage Intel's SA-1 StrongARM pipeline [3] 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

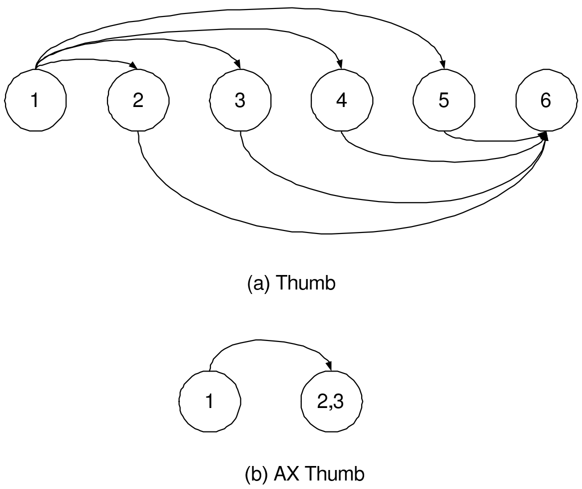


Figure 11: SetAllHigh AX transformation

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 `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` [7],`Commbench` [14] and `Net-Bench` [8] 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`.

Instruction Counts The use of AX instructions reduces the dynamic instruction count of 16-bit code by 0.4% to 32%. Figure 12 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 perfromance gap between 32-bit and 16-bit code. For cases such as `crc` and `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 a factor of 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 13 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 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.

Code Size and Fetch Data 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 14. Figure 15 shows the I-cache fetches for Thumb and AXThumb codes relative to ARM code. In the three cases where Thumb has more I-cache fetches viz. `crc` and the two `adpcm` programs, we see that AXThumb reduces the fetches making them almost as little as ARM. In the other cases we see AX always has fewer I-cache fetches compared to Thumb, making it even better compared to ARM. Fewer fetches could result from code size reducing AX transformations. 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 instruction that are flushed, which account for the extra fetches performed by Thumb. From an energy perspective, we see that energy spent on the I-cache will be lesser in AXThumb compared to Thumb. Additionally, since the instruction count is reduced, energy spent 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. Hence we also save on overall energy using AX instructions.

Usage of AX instructions In Table 1 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`, `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.

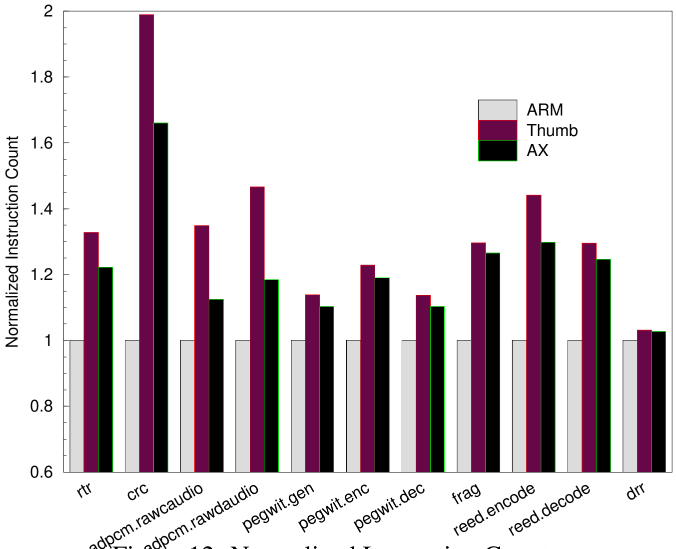


Figure 12: Normalized Instruction Counts

4 Conclusions

The design of dual instruction width processors like ARM poses an important challenge. Some of the functionality of the 32 bit ARM instructions must be sacrificed to obtain a more compact 16 bit encoding for Thumb instructions. We have demonstrated an approach which very effectively compensates for the weaknesses of the 16-bit code bridging the performance gap between 16-bit and 32-bit codes without detriment to the code size and energy reducing properties of 16-bit code. A new class of AX instructions is carefully designed so that extra Thumb instructions can be eliminated at runtime through instruction coalescing performed in the processor's decode stage. These instructions were implemented using exactly one unused opcode in the 16-bit encoding space. The compiler is responsible for identifying Thumb instructions that can be eliminated and replacing them with appropriate AX instructions. The hardware extensions are simple and by handling the AX instructions in parallel with other instructions we avoid any increase in the processor's cycle time.

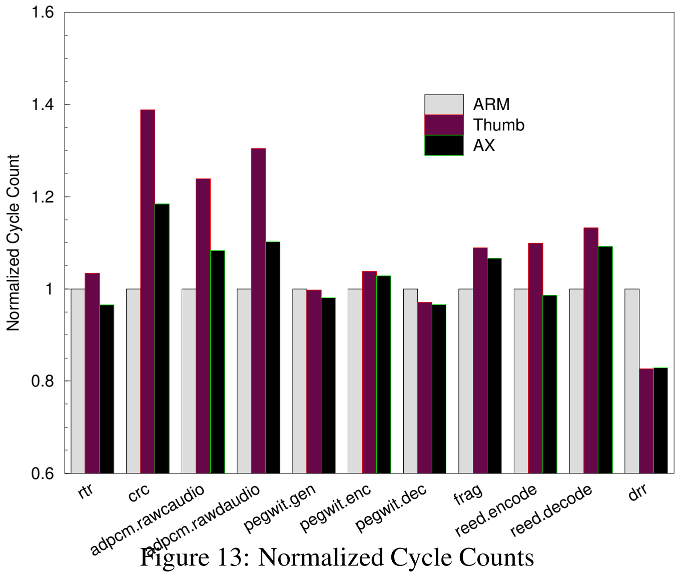


Figure 13: Normalized Cycle Counts

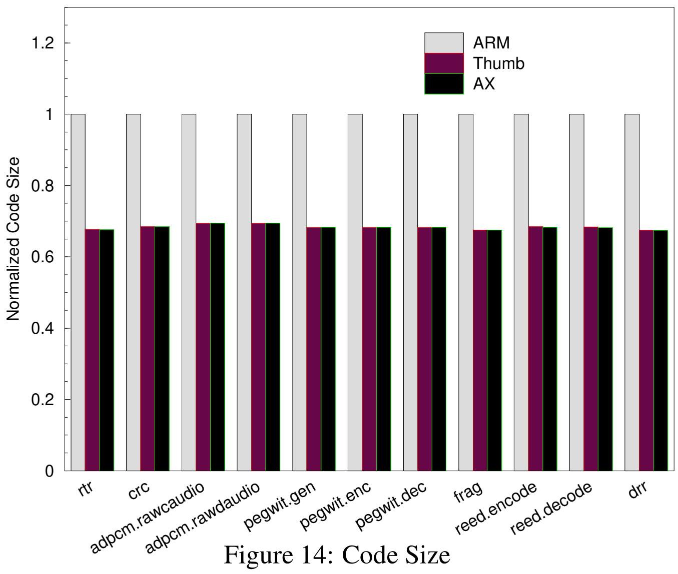


Figure 14: Code Size

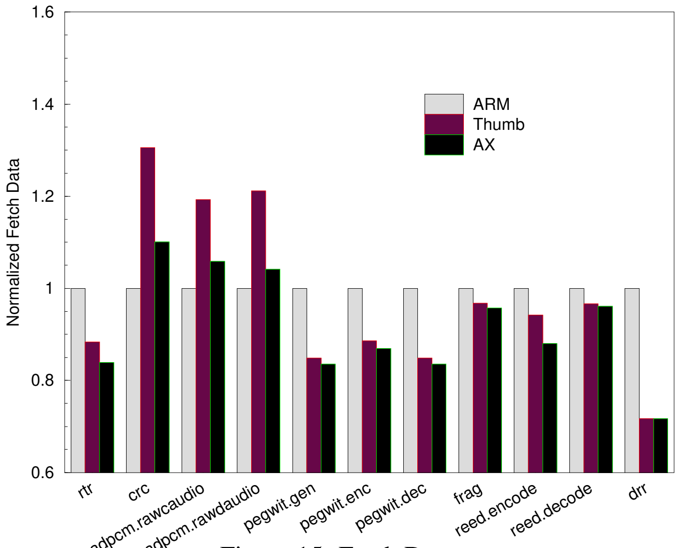


Figure 15: Fetch Data

Table 1: 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%

References

[1] D. Burger and T.M. Austin, "The Simplescalar Tool Set, Version 2.0," *Computer Architecture News*, pages 13–25, June 1997.

[2] S. Furber, "ARM system Architecture," Publisher: Addison Wesley Longman, 1996.

[3] Intel Corporation, "SA-110 Microprocessor Technical Reference Manual"

[4] Intel Corporation, "The Intel XScale Microarchitecture Technical Summary"

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

[6] Removed to preserve authors anonymity.

[7] C. Lee, M. Potkonjak, and W.H. Mangione-Smith, "Mediabench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," *IEEE/ACM International Symposium on Microarchitecture* (MICRO), Research Triangle Park, North Carolina, December 1997.

[8] Memik, Mangione Smith and Hu, "NetBench: A Benchmarking Suite for Network Processors," *IEEE International Conference on Computer-Aided Design*, November 2001

[9] MIPS Technologies, "MIPS32 Architecture for Programmers Volume IV-a: The MIPS16 Application Specific Extension to the MIPS32 Architecture," March 2001.

[10] J. Montanaro et al., "A 160-MHz, 32-b, 0.5-W CMOS RISC Microprocessor," *IEEE Journal of Solid-State Circuits*, Vol. 31, No. 11, November 1996.

[11] G. Reinman and N. Jouppi, "An Integrated Cache Timing and Power Model," *Technique Report*, Western Research Lab., 1999.

[12] D. Seal, Editor, "ARM Architecture Reference Manual," Second Addition, Addison-Wesley.

[13] S. Wilton and N.Jouppi, "An Enhanced Access and Cycle Time Model for On-Chip Caches," *Technique Report*, Western Research Lab., May 93.

[14] T. Wolf and M. Franklin, "Commbench - A Telecommunications Benchmark for Network Processors," *IEEE International Symposium on Performance Analysis of Systems and Software*, April 2000.