Enhancing the Performance of 16-bit Code Using Augmenting Instructions *

Arvind Krishnaswamy Rajiv Gupta

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

ABSTRACT

In the embedded domain, memory usage and energy consumption are critical constraints. Dual width instruction set embedded processors such as the ARM provide a 16-bit instruction set 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. We have observed that throughout 16-bit Thumb code there exist Thumb instruction pairs that are equivalent to a single ARM instruction. We have developed an approach which uses combination of compiler and architectural support to exploit the above property for improving performance of 16-bit code. We enhance the Thumb instruction set by incorporating *Augmenting eXtensions* (AX). The task of the compiler is to identify pairs of Thumb instructions that can be safely combined and executed as single ARM instructions. The compiler replaces such pairs of Thumb instructions by AX+Thumb instruction pairs. The AX instruction is coalesced with the immediately following Thumb instruction to generate a single ARM instruction at decode time. Thus, using AX instructions, the compiler can both generate compact 16-bit code and provide hardware with information needed to produce better performing 32-bit code.

Categories and Subject Descriptors

C.1 [**Computer Systems Organization**]: Processor Architectures; D.3.4 [**Programming Languages**]: Processors—*compilers*

General Terms

Algorithms, Measurement, Performance

Keywords

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

*Supported by NSF grants CCR-0220334, CCR-0208756, CCR-0105355, and EIA-0080123 to the University of Arizona.

LCTES'03, June 11–13, 2003, San Diego, California, USA.

1. INTRODUCTION

More than 98% of all microprocessors are used in embedded products, the most popular 32-bit processor 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, applications must execute under constraints of limited memory and low energy consumption. As complex applications, such as image processing and graphics intensive games, are being ported to embedded platforms, performance or speed of execution is becoming equally important. Dual instruction set processors, such as the ARM, MIPS16 [9], SuperH-5 [13], address the limited memory/energy constraint 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. While the 16-bit code expends lesser energy and has a smaller memory footprint, it spends many more cycles in execution time.

Traditionally, ISAs have been fixed width (e.g., 32-bit SPARC, 64-bit Alpha) or variable width (e.g., x86). Fixed width ISAs give good performance at the cost of code size and variable width ISAs give good performance at the cost of added decode complexity. Neither of the above are good choices for embedded processors where code size and power are critical. Dual width ISAs are simple to implement and provide a tradeoff between code size and performance, making them a good choice for embedded processors. We propose an extension to the 16-bit ISA that serves as a bridging ISA in dual width processors. We call this extension `Augmenting eXtensions` or AX.

AX instructions are non-executing instructions that do not contribute to execution time. An AX instruction is coalesced with the following 16-bit Thumb instruction at decode time. Since AX instructions are also 16-bit instructions, they have the energy saving and small code size properties of Thumb code. AX instructions, unlike prefix instructions found in architectures such as the MIPS16 [9] and SuperH-5 [13], do not merely improve the expressibility of 16-bit code; they do so without adding any cycles to the execution time via instruction coalescing. Instruction Coalescing is a scalable technique that improves the performance of 16-bit code making it possible to bridge the performance gap between 32-bit ARM and 16-bit Thumb code. While the AX extensions described in this paper are for the ARM architecture, the idea of Instruction Coalescing and Augmenting instructions can be applied to other dual width processors. In previous work [6] we showed how one could achieve good code size, low energy and high performance using profile guided heuristics at compile time. The techniques described here are orthogonal to the previous techniques and more scalable.

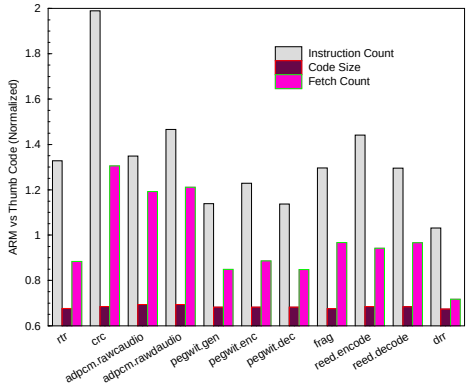
In this paper we describe the microarchitecture and compiler support used to take advantage of AX instructions.

The remainder of the paper is organized as follows. Section 2 gives an overview of the ARM architecture with a comparison between 32-bit ARM code and 16-bit Thumb code. Section 3 gives an overview of Instruction Coalescing, including a description of the enhanced microarchitecture, a description of predication support and a brief description of the AX extensions. Section 4 describes in detail the 3 phases used by the compiler postpass to transform Thumb code into AXThumb code. We present the results in Section 5 and conclude in Section 6.

2. BACKGROUND

The ARM architecture is a dual width RISC architecture supporting a 16-bit and 32-bit ISA. The processor is said to be in the ARM state when executing 32-bit instructions and Thumb state when executing 16-bit instructions. The 16-bit ISA in any dual width processor can only capture part of expressibility and functionality of the 32-bit ISA. The goal of Augmenting Instructions is to make up for this lost expressibility and functionality without paying the price of execution cycles. Some of the important differences between the ARM and Thumb instruction sets are as follows. Most Thumb instructions cannot be predicated while ARM supports full predication. Most Thumb instructions use a 2-address format (destination register is the same as the first source) while ARM supports 3-address format for manipulating 32 bit data. Visible registers in Thumb state are `r0` through `r7`; only some instructions, mainly MOV and ADD instructions, can directly address registers `r8` through `r15`. In ARM state all sixteen registers from `r0` through `r15` are visible across various instructions. The *Branch and Exchange Instruction* (BX) instruction can be used to switch between ARM and Thumb states.

Figure 1 ARM vs 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] we have 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.

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 Section 4. 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 state and 32-bit state for short sequences negates the benefit of mixed code, as is shown in Appendix A.

3. INSTRUCTION COALESCING

Instruction Coalescing is first introduced using an example. The idea of AX extensions and a description of the microarchitecural extensions required for AX processing is given. The support of predication using AX instructions is explained next.

3.1 Basic Idea

```
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 Instruction Coalescing 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 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. 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.

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 AX-Thumb 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.

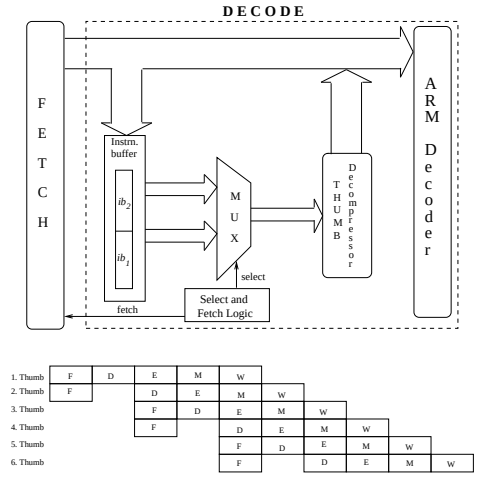
3.2 Microarchitecture

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. The changes described here are entirely in the decode stage. Most dual width embedded processors are simple pipelined machines, making instruction coalescing easily implementable. Thus, the techniques described here are not restricted to the ARM family of processors.

3.2.1 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 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* 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, typically fetch needs to be carried out in alternate cycles.

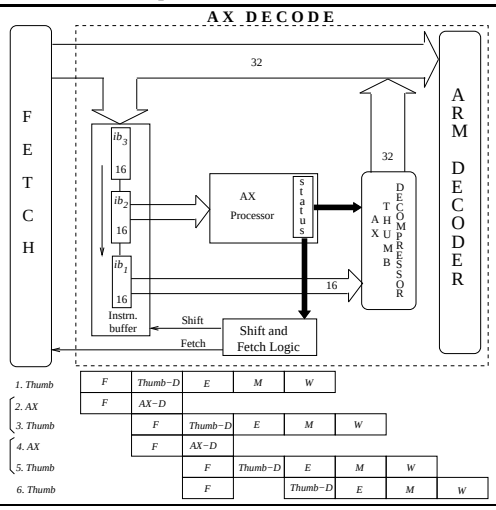
Figure 2 Thumb Implementation.



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.

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. During a thread switch it is sufficient to save the state of the status register along with other state to ensure correct execution when thread resumes execution.

Figure 3 AXThumb Implementation.



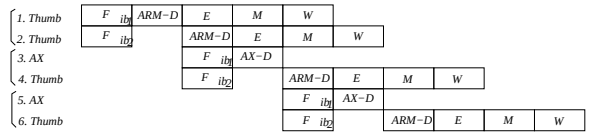
There are three important points to note about the above operation. First, as shown by the pipeline timing diagram in Figure 3, in the above operation *no extra cycles* are needed to handle the AX instructions. Each sequence (pair) of AX and Thumb instructions complete their execution one cycle after 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. Third, it should 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.

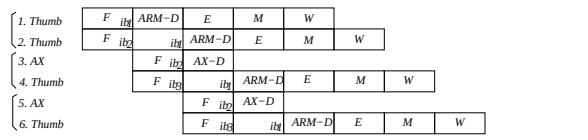
The instruction buffer and the filling of this buffer by the instruction fetch mechanism are designed such that, in the absence of taken branches, the instruction buffer always contains at least two instructions. The buffer can hold up to three consecutive instructions. Thus, it is expanded in size from 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.

Figure 4 Delivering Instructions to Decode Ahead for Overlapped Execution.



(a) 32 bit Instruction Buffer.



(b) 48 bit Instruction Buffer.

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 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 is 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 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.

3.2.2 Predicated Execution in AXThumb

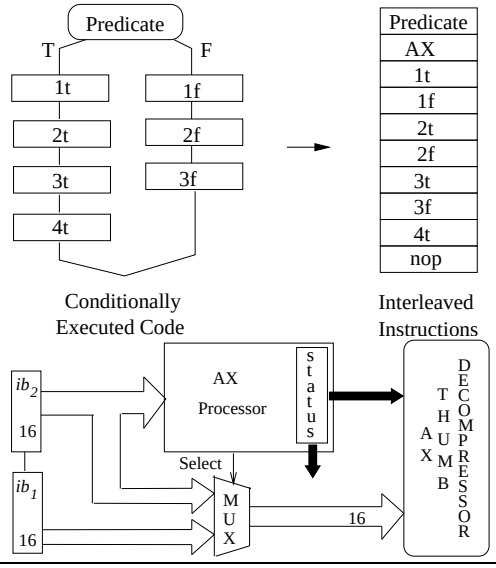
While the original Thumb instruction set does not support predicated execution, we have developed a very effective approach to carry out predicated execution using AXThumb code which requires only a minor modification to the decode stage design just presented. Like instruction coalescing, this method 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 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

[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.

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.

Figure 5 Predication in AXThumb.



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 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.

3.3 Augmenting eXtensions

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. We show how we use exactly one free instruction in the free opcode space of the Thumb instruction set to implement 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]

Table 1: AX Instructions

AX Instruction	Description
`setpred`	support for predication in 16-bit code
`setsbit`	sets the 'S' bit to avoid explicit `cmp` instructions
`setsource`	sets the source register for the next instruction
`setdest`	sets the destination register for the next instruction
`setthird`	sets the third operand (support 3-address format)
`setimm`	sets the immediate value for the next instruction
`setshift`	sets the shift type and amount for the next instruction
`setallhigh`	indicates next instruction uses all high registers

A short description of the AX extensions can be found in Table 1. 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` is 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]

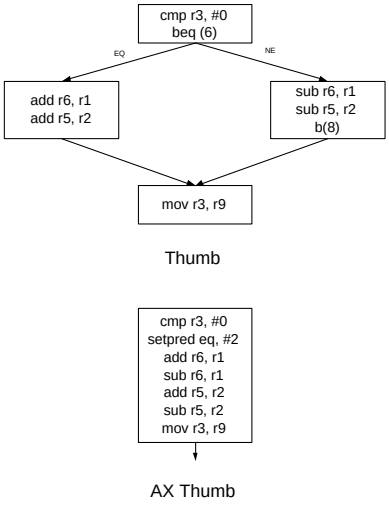
101110	setthird	reg	-
[10..15]	[7..9]	[3..6]	[0..2]

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

Figure 6 Predication.



4.1 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 in each pair is executed.

Original ARM
`cmp r3, #0`
`addeq r6, r6, r1`
`addeq r5, r5, r2`
`rsbne r6, r6, r1`
`rsbne r5, r5, r2`
`mov r3, r9`

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

Corresponding 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`

Coalesced ARM
`cmp r3, #0`
`sub r6, r6, r1`
`sub r5, r5, r2`
`mov r3, r9`
OR
`cmp r3, #0`
`add r6, r6, r1`
`add r5, r5, r2`
`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 form of predication could also reduce the number of fetches from the I-cache. The example shown below illustrates one such case. In the case of Thumb code, the instructions immediately following the branch instructions will be fetched even on taken branches, resulting in wasted fetches. In the AXThumb case, for every pair of instructions that is fetched at least one instruction is executed, making all fetched pairs useful. Also note that the use of predication reduces the size by one instruction in this case.

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

AXThumb
`L0: I0`
`setpred EQ 1`
`I1`
`I2`
`beq L0`

Figure 7: SetPredicate

input : A CFG for a function
output : A modified CFG with 'set'predicated code
for *all siblings* (n_1, n_2) *in the BFS Traversal of the CFG* **do**
  /* Check for a hammock in the CFG */
  $PredEQ$ = $SuccEQ$ = FALSE;
  if `numPreds` (n_1) == `numPreds` (n_2) == *1* **then**
    if `Pred` (n_1) == `Pred` (n_2) **then**
      $PredEQ$ = TRUE;
    end
  end
  if `numSuccs` (n_1) == `numSuccs` (n_2) == *1* **then**
    if `Succ` (n_1) == `Succ` (n_2) **then**
      $SuccEQ$ = TRUE;
    end
  end
  /* SetPredicate if hammock found */
  if $SuccEQ$ *and* $PredEQ$ **then**
    `DeleteLastIns(` `Pred(`n_1`)``)`;
    `InsertIns(` `Pred(`n_1`)`, *setpred, cond* `)`;
    for *each pair of instructions* in_1, in_2 *from* n_1 *and* n_2
    do
      `InsertIns(` `Pred(`n_1`)`, in_1 `)`;
      `InsertIns(` `Pred(`n_1`)`, in_2 `)`;
    end
    `MergeBB(`*Pred*(n_1), *Succ*(n_1)`)`;
    `DeleteBB(`n_1`)`;
    `DeleteBB(`n_2`)`;
  end
end

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 in Figure 7, 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 7. The CFGs for the code example described above, before and after the transformation are shown in Figure 6.

4.2 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.

Original ARM

mov r2, r5
sub r1, r2, lsl #5
ldr r5, [r9, #100]

Corresponding 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]

Coalesced ARM

mov r2, r5
sub r1, r2, lsl #5
ldr r5, [r9, #100]

Figure 8: DAG Coalescing for generic AX instructions

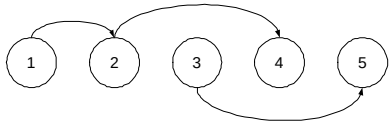
input : Basic Block DAG D with nodes numbered according to the topological order and register liveness information
output : Basic Block DAG D with Coalesced Nodes to indicate AXThumb instruction pairs

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

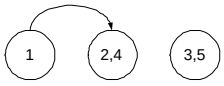
The algorithm shown in Figure 8 uses the Basic Block dependence DAG and global register liveness information as its input. Since AXThumb pairs replace dependent Thumb instructions, it is sufficient to examine adjacent nodes along a path in the DAG. We traverse the DAG in Breadth-First Order and examine each node and its predecessor. AXThumb pairs have to be instructions adjacent to each other in the instruction schedule. While replacing Thumb pairs with equivalent AXThumb pairs, 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 sort 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.

The DAG for our example, before and after the transformation is shown in Figure 9. For this 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.

Figure 9 Phase 2



(a) Thumb



(b) AX Thumb

We describe some more examples of AXThumb transformations that use this algorithm below.

Avoiding 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 use 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

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

We describe one more scenario where the setshift instruction can be used. 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 state 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`

High Register Operands. Analogous to the `setsource Hreg` instruction that was introduced earlier, `setdest Hreg` instruction causes the Thumb instruction following the `setdest Hreg` instruction to use `Hreg` as its destination register. The instruction following coalescing of AXThumb instructions is identical to the corresponding ARM instruction.

Original ARM
`ldr Hreg, [reg, #offset]`

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

AXThumb
`setdest Hreg` `ldr _, [reg, #offset]`

Coalesced ARM
`ldr Hreg, [reg, #offset]`

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`

4.3 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 state, 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.

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

Corresponding Thumb

`(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 [r0, r1, r2, r3]`

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

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 10. 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 11. The `PushorPopList` functions find instructions that push/pop a list of 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 the two instructions are adjacent to each other in the instruction schedule.

Figure 10 SetAllHigh AX transformation

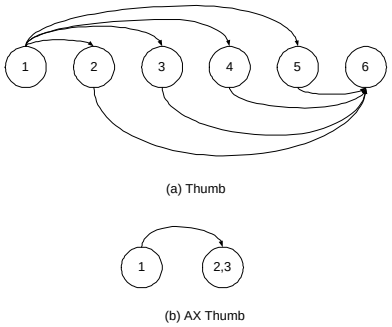


Figure 11: DAG Coalescing for setallhigh AX instructions

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 ∈ set of basic blocks B do
    for each n ∈ BFS order of nodes in D do
        if PushOrPopListLo(n) then
            /* Check for the replaceable mov instructions */
            isReplacable = TRUE
            for each m ∈ 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 ∈ Succ(n) do
                    Save ← Succ(m)
                    Remove(m)
                end
                Succ(n) ← Save
                SettoLo(Save)
                Coalesce(setallhigh, Succ(n))
            end
        end
    end
end
```

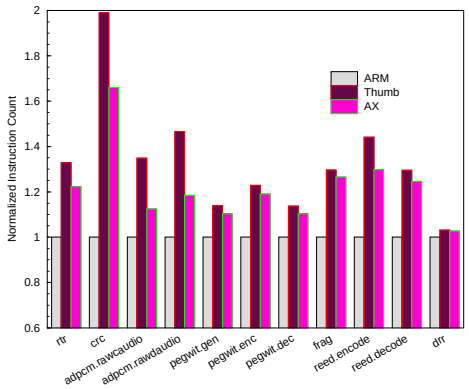
5. EXPERIMENTAL RESULTS

Methodology. The algorithms described above were used to detect candidate Thumb instructions in the assembly level code of the most frequently executed functions in the benchmark programs. 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 transformations were then carried by transforming the assembly code. The modified assembly code was then linked in with the rest of the code.

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 64 cycles (a miss requires going off-chip). The simulator was extended to support both 16-bit and 32-bit states, 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` [7],`Commbench` [12] and `NetBench` [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 performance 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.

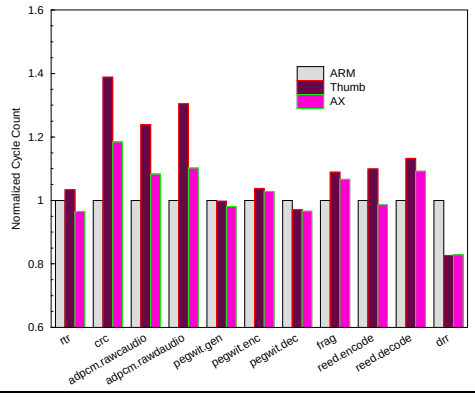
Figure 12 Normalized Instruction Counts



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.

Figure 13 Normalized Cycle Counts



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 14 Code Size

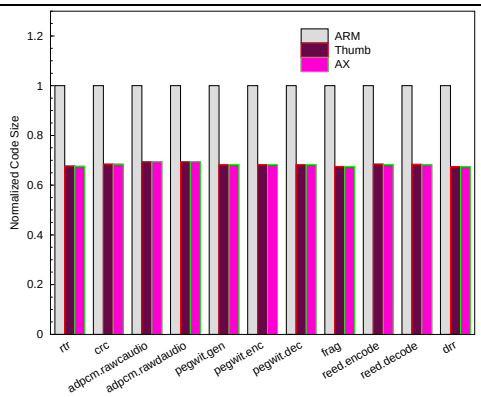
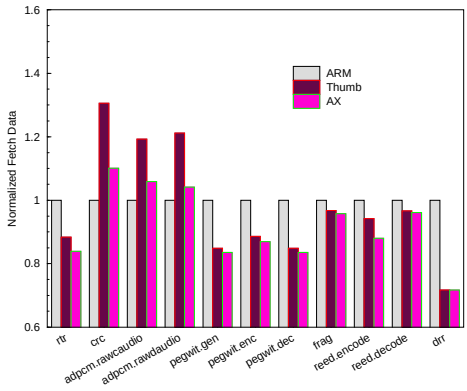


Figure 15 shows the I-cache fetches for Thumb and AXThumb codes relative to ARM code. In the three cases where Thumb has more I-cache fetches viz. `crc` and the two `adpcm` programs, we see that AXThumb reduces the fetches making them almost as little as ARM. In the other cases we see AX always has fewer 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 of instructions fetched 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. 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.

Figure 15 Fetch Data



Usage of AX instructions. In Table 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`, `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.

6. CONCLUSIONS

We proposed the concept of Instruction Coalescing using a set of Augmenting eXtensions to the existing 16-bit code. The AX instructions are a bridging ISA between the existing 16-bit and 32-bit ISAs in dual width processors capable of making the performance of 16-bit code close to that of 32-bit code. AX instructions improve the expressibility of 16-bit code without adding to the cost in terms of execution cycles. We presented algorithms used to generate such 16-bit AXThumb code from existing 16-bit Thumb code. The results show that using just 8 AX instructions we are able to considerably improve performance of 16-bit code without negatively affecting code size and I-cache fetches. While the techniques described here were implemented in the context of the ARM Architecture [11], they can be applied to other dual width embedded processors. Using the compiler algorithms described here and devoting more encoding space to AX type of bridging ISAs, one could further bridge the performance gap between 32-bit and 16-bit code.

Table 2: Usage of Different AX Instructions.

Benchmark	setallhigh	setpred	setsbit	setshift
rtr	11.77%	0.00%	82.34%	5.88%
crc	0.00%	0.00%	0.27%	99.72%
adpcm.rawcaudio	0.00%	36.30%	36.30%	14.52%
adpcm.rawdaudio	0.00%	34.47%	34.47%	13.79%
pegwit.gen	0.17%	0.00%	74.47%	8.48%
pegwit.encrypt	0.19%	0.00%	80.22%	5.01%
pegwit.decrypt	0.17%	0.00%	74.47%	8.48%
frag	4.44%	0.00%	0.00%	6.66%
reed.encode	0.01%	0.00%	3.81%	0.00%
reed.decode	0.01%	0.00%	1.09%	0.63%
drr	0.00%	0.00%	100.00%	0.00%

Benchmark	setsource	setdest	setthird	setimm
rtr	0.00%	0.00%	0.00%	0.00%
crc	0.00%	0.00%	0.00%	0.00%
adpcm.rawcaudio	0.00%	7.26%	0.00%	5.59%
adpcm.rawdaudio	3.44%	10.34%	3.44%	0.00%
pegwit.gen	5.47%	0.00%	11.39%	0.00%
pegwit.encrypt	6.23%	0.00%	8.32%	0.00%
pegwit.decrypt	5.47%	0.00%	11.39%	0.00%
frag	13.33%	4.44%	66.66%	4.44%
reed.encode	68.45%	0.00%	27.71%	0.00%
reed.decode	88.29%	0.00%	9.95%	0.00%
drr	0.00%	0.00%	0.00%	0.00%

7. 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 Core Developer's Manual"

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

[6] A. Krishnaswamy and R. Gupta, "Profile Guided Selection of ARM and Thumb Instructions," *ACM SIGPLAN Joint Conference on Languages Compilers and Tools for Embedded Systems & Software and Compilers for Embedded Systems* (LCTES/SCOPES), Berlin, Germany, June 2002.

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

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

[13] Kunio Uchiyama, "The SH-5/ST50: An Advanced Microprocessor Core for Networking and Multimedia Applications," *Cool Chips III*, April 2000.

Appendix A

Illustration of switching states using Branch and eXchange.

When executing ARM instructions, the execution of `BX Rm` instruction can be used to begin executing Thumb instructions. Other dual width processorss have a similar mechanism to switch execution states. `BX Rm` has the following semantics. If bit `Rm[0]` is 1, the processor switches to execute Thumb instructions. It begins executing at the address in `Rm` aligned to a half-word boundary by clearing the bottom bit. If bit `Rm[0]` is 0 then the processor continues to execute ARM instructions, that is, `BX` simply behaves as a branch instruction in this case. The current state in which the processor is executing is indicated by the `T` bit which is bit 5 of the `CPSR` (Current Program Status Register). This bit is appropriately changed when the processor state is switched. Similarly the `BX` instruction can be used to switch from Thumb state to ARM state.

The use of `BX` instruction to generate a mixed binary is shown in Figure 3. 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. Therefore only if the shorter ARM sequence contains greater than four fewer instructions than the longer Thumb sequence, the generation of mixed binary is beneficial.

Thumb	
`.code 16`	; Thumb instructions follow
...	
<Longer Thumb Sequence>	
...	

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
<Shorter ARM Sequence>	;
`orr r15, r15, #1`	; set r15[0]
`bx r15`	; switch to Thumb as r15[0] is set
`.code 16`	; Thumb instructions follow
...	

Table 3: Replacing Thumb Sequence by ARM Sequence.