By Arvind Krishnaswamy and Rajiv Gupta

MIXED-WIDTH INSTRUCTION SETS

Encoding a program's computations to reduce memory and power consumption without sacrificing performance.

Applications written for the embedded domain must perform under the constraints of limited memory and limited energy. While these constraints have always existed, current trends, such as mobile computing and ubiquitous computing, bring more and more complex applications to the embedded domain, making performance, or speed of execution, an important factor as well. For instance, we are now able to run resource-intensive gaming and multimedia applications on handheld devices. Techniques that reduce the memory and energy consumption of programs have in general done so at the cost of performance. Simultaneously achieving small code size, low energy consumption, and high performance is a challenging task.

Processors, such as the ARM and MIPS family of embedded cores, support more than one instruction set to meet these constraints. In addition to the 32-bit instruction set, they support a 16-bit instruction set. As we will explain, by using 16-bit code one can achieve code size and energy reduction at the cost of performance.

Until recently, the choice between 32-bit and 16-bit code had to be made by the programmer, and is highly undesirable. In this article, we show how this task can be automated and how one can achieve the code-size and energy-saving properties of 16-bit code while simultaneously achieving performance comparable to 32-bit code. We focus on the encoding of a program's computations. This is in contrast to the previous articles in this special section, which primarily focus on the elimination of superfluous computations from a program.

The techniques described here are in the context of the ARM family of processors, which are frequently used in the embedded computing realm. They are used as general-purpose embedded processors, found, for example, on multimedia-enabled PDAs, as well as in specialized embedded applications, such as embedded control. ARM processors have a simple energy-efficient architecture. The StrongARM [8],

for example, has a five-stage pipeline. It performs in-order issue, and has no branch prediction, has a 32-kilobyte instruction cache and 32-kilobyte data cache. Recent ARM processors, such as the Xscale [2], still maintain shallow pipelines but incorporate branch prediction and out-of-order completion. Unlike high-performance processors, these processors have simple architectures, since they have a very tight energy and cost budget.

WHEN performance is an important criterion, 32-bit ARM code should be the choice; when code size is an important constraint, 16-bit Thumb code should be the choice.

A Closer Look at the ARM Architecture

The ARM is a 32-bit RISC architecture [10] supporting two instruction sets, a 32-bit ARM instruction set, and a 16-bit Thumb instruction set. Corresponding to the two instruction sets are two execution states. In ARM state, 32-bit instructions are executed, and in Thumb state, 16-bit instructions are executed. The ARM ISA (instruction set architecture) supports a three-address format, predicated execution, and can access all sixteen 32-bit registers. In Thumb state, however, instructions are restricted to a two-address format, can access only eight 32-bit registers in most cases, and do not support predicated execution. The Thumb instruction set has limited expressive power compared to the ARM instruction set. For example, in an ARM instruction it is possible to specify a shift operation along with an ALU operation in the same 32-bit instruction, but in Thumb state two instructions are required.

Since the full expressiveness of the 32-bit ARM ISA is not always necessary, one can achieve considerable code size reductions using 16-bit Thumb instructions. The Thumb version of an application is on average 30% smaller than its 32-bit ARM counterpart [4]. It should be noted that using Thumb code, with every 32 bits fetched, the processor fetches two Thumb instructions. Hence the processor needs to fetch a word only every other cycle, reducing the amount of energy spent on fetching instructions from the instruction cache. Considering that a lot of energy is spent in the instruction cache (the cache is fully associative, requiring multiple simultaneous lookups), this reduction is significant. Thumb code being small also provides a good locality of reference. Therefore there are fewer cache misses in Thumb code compared to ARM code. While there is a significant reduction in code size and energy when we use Thumb code, sometimes we lose a considerable amount of performance. This is because for the same task we need many more Thumb instructions compared to the number of

ARM instructions. This loss is incurred in spite of the good locality provided by Thumb code. We have observed this performance loss to vary between 3% and 98%. From the preceding discussion it is clear that when performance is an important criterion, 32-bit ARM code should be the choice; when code size is an important constraint, 16-bit Thumb code should be the choice.

The ARM architecture supports switching between these instruction sets within a single executable. The Branch and Exchange (`BX`) instruction and Branch with Link and Exchange (`BLX`) are provided to perform this switch. The operand used by these instructions is a register operand. When the register operand has its least significant bit set, the processor continues execution in Thumb state, and execution continues in ARM state when the bit is unset. The upper 31 bits of the effective address specify the branch target.

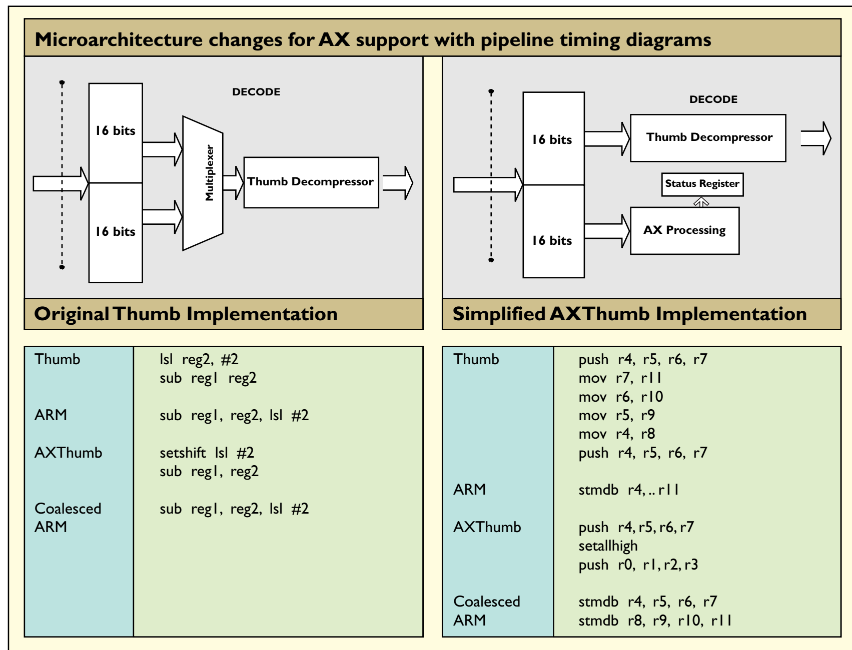

Figure 1. The adaptation of the pipeline in order to execute AXThumb code, and the two code fragments in Thumb, ARM, and AXThumb and their coalesced equivalent. Note that the coalesced ARM instruction is composed from an AXThumb instruction pair at decompressing time and fed to the ARM decoder.

The `BX` instruction can be used by assembly-level programmers to switch between the two states. Recent compilers also support directives that allow the programmer to specify the state for a given compilation unit. Both preceding approaches require the programmer to decide where to switch states. With more complex applications being ported to the embedded platform, this burden on the programmer becomes unreasonable, making an automatic approach more desirable.

Instr.	use
setthird	overcomes the lack of 3-address format
setimm	allows to specify large immediate constants
setsource	specifies the source operand of the following instruction
setdest	specifies the destination operand of the following instruction
setshift	specifies the shift operation and shift amount to be applied
setsbit	sets the 'S' bit to avoid the explicit use of compare instructions
setallhigh	indicates that the next instruction uses only the high registers
setpred	used to support predication

AXThumb additional instructions.

Profile-Guided Generation of Mixed Code

The basic approach we take for automatically generating mixed code consists of two steps. First, we find the frequently executed functions using profiling. Each of these are functions takes up more than 5% of total execution time. Second, we use a heuristic for choosing between ARM and Thumb codes for these frequently executed functions. For all other functions, we generate Thumb code. The preceding approach is based upon the observation that we should use Thumb state whenever possible. Functions for which the use of Thumb code results in significantly lower overall performance must be compiled into ARM code.

In order to decide between the use of ARM code and Thumb code for a frequently executed function, we essentially compare the characteristics of the ARM and Thumb code for that function. We make the final decision based upon the expected performance of the two versions of the functions (ARM and Thumb) and their relative code sizes.

We use a combination of dynamic instruction counts and relative code sizes to make the decision. In particular we choose Thumb code if one of the following conditions hold: the Thumb instruction count is lower than the ARM instruction count, or the Thumb instruction count is higher by no more than `T1`% and the Thumb code size is smaller by at least `T2`%.

The idea behind this heuristic is that if the Thumb instruction count for a function is slightly higher than the ARM instruction count, it may still be appropriate to use Thumb code if it is sufficiently

smaller than the ARM code as the smaller size may lead to fewer instruction cache accesses and misses for the Thumb code. The net effect is intended to be that the cycle count of Thumb code should not be higher than the cycle count for the ARM code.

We found that the Mixed code size is significantly smaller than the ARM code size and only slightly bigger than the Thumb code size. We also observed that Mixed code gives instruction cache energy savings over the ARM code. Moreover, the energy savings are comparable to those obtained by Thumb code. Finally, the cycle count of Mixed code is very close to the cycle count of the ARM code. In some cases it is even slightly smaller due to the improved instruction cache behavior of Mixed Code. A more detailed analysis, including a comparison with three other heuristics, is available in [4].

Switching States at Finer Granularity

The profile-guided approach was applied at the granularity of functions. Each function was compiled entirely into ARM code or Thumb code. This is because switching states is practical only when applied to long sequences of instructions. State switch is achieved using the BX instruction. The problem is that the overhead involved in switching states often overcomes the benefit of switching when switching at finer granularities, such as at the instruction level. This overhead consists not only of the switching instruction but usually also involves a number of no-ops to ensure such things as correct alignment and correct usage of the BX instruction. This overhead would negate the benefit of replacing a small Thumb sequence with an ARM code sequence. Using the profile-guided approach at finer granularities is hence wasteful.

While the switch is useful only for large code sequences, using a peephole scan, one can find many instances of short 16-bit sequences that have faster ARM equivalents. One such case is illustrated in the lower left half of Figure 1. The task is to shift the contents of reg1 before subtracting from reg2. In Thumb state, the shift operation requires a separate instruction. It cannot be specified along with the subtract instruction due to the lack of encoding space (we have only 16 bits). In 32-bit ARM state, however, both the shift and subtract can be specified in one single cycle instruction. While both the ARM and Thumb code are 32 bits long, the ARM code is faster, and therefore desirable where performance is critical.

In order to be able to avoid the slowdown of Thumb compared to ARM in such cases, we propose to extend the Thumb instruction set to accommodate coalescable instructions called Augmenting Extensions (AX). The processor pipeline is modified to implement Instruction Coalescing using these instructions. Using these AX instructions one can generate 16-bit AXThumb code, which has higher performance compared to Thumb code while retaining the code size and energy-saving properties of Thumb code.

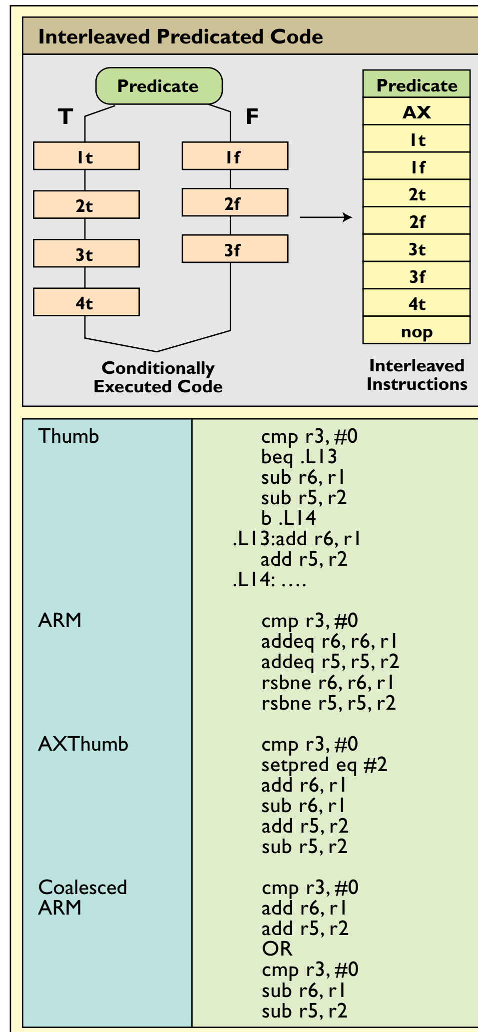

Figure 2. Predication support in AXThumb and a code fragment showing the effect of predication.

Instruction Coalescing and AXThumb

The idea is to coalesce two 16-bit instructions and execute one instruction instead. To this end, we extend the 16-bit Thumb instruction set with augmenting instructions to enable the processor to perform coalescing. The decode stage of the processor is modified to enable two 16-bit Thumb instructions to be coalesced at runtime. The compiler generates code using these AXThumb extensions replacing pairs of Thumb instructions with AXThumb instructions. Unlike other prefix instructions (used in [7] for example) where the purpose of the prefix instructions was to improve the expressive power of 16-bit instructions, AXThumb is an extension to the ISA and microar-

chitecture, enabling the execution of these instructions at zero cost in terms of execution cycles.

The AX extension to the Thumb instruction set consists of eight instructions, each addressing a particular limitation of the Thumb instruction set. An overview of these instructions is given in the table here. These AX extensions essentially carry some information required for the correct execution of the instruction that immediately follows it. These instructions are processed entirely in the decode stage of the processor without going through the rest of the pipeline. The decode stage is modified to examine two consecutive 16-bit instructions to allow for the AX processing to be carried in parallel with the execution of another 16-bit instruction, as shown in the upper half of Figure 1. In case an AX instruction is found by the AX processing unit, the relevant state carried by the instruction is saved in a special state register. The contents of this register are then used when the next 16-bit instruction is decoded. Hence, the overall effect is like coalescing two 16-bit instructions whenever possible, executing only one instruction.

We describe two examples as shown in the lower half of Figure 1, to illustrate the benefits of Instruction Coalescing. First, we revisit the shift example described earlier. While the ARM version requires the execution of one instruction, the Thumb version requires the execution of two instructions. This is overcome in AXThumb where the `setshift` instruction is coalesced with the following `sub` instruction, thereby executing only one instruction, shown as the Coalesced ARM instruction previously. This coalescing is a zero-cost operation in terms of cycles, since the coalescing is done in parallel with the processing of a previous instruction. Hence, we save one cycle for every AX Thumb pair in comparison to a Thumb pair. In essence we have 16-bit code that can perform like 32-bit code.

The second example uses the `setallhigh` instruction. The call semantics in the ARM architecture require the callee to save and restore nonvolatile registers. In Thumb state, saving is done using the push instruction, which pushes the contents of a list of registers onto the stack. The push instruction can specify only the low registers requiring a set of moves to move the contents of high registers to the low registers before they can be saved. This is avoided using AXThumb `setallhigh` instruction, as illustrated in the example code in the right lower half of Figure 1.

A detailed description of Instruction Coalescing and the AX Extensions can be found in [3].

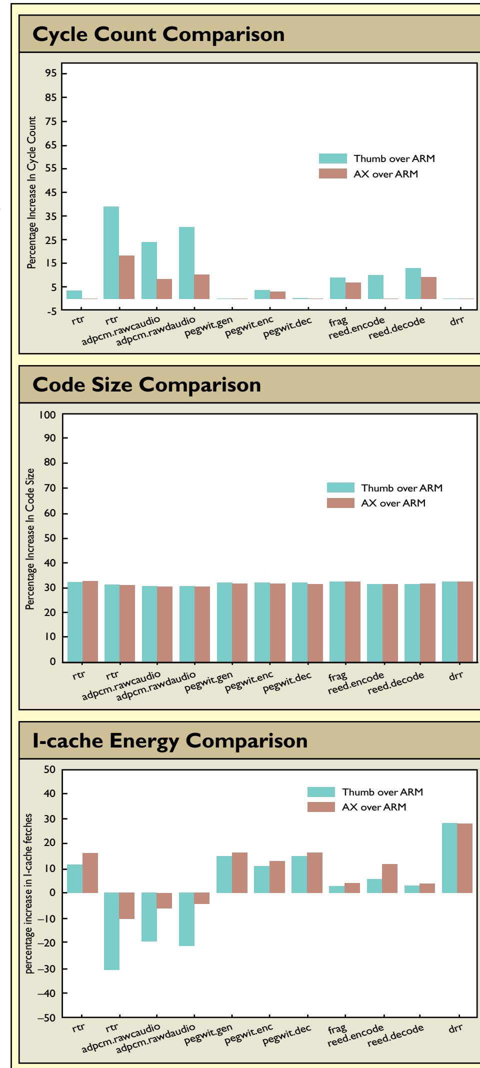

Figure 3. Results: Cycle count, code size, and instruction cache fetch energy.

Predication

Using AXThumb we also support predication in Thumb state. Like instruction coalescing, this method also takes advantage of the extra fetch bandwidth (32 bits) already present in the processor. We rely on the compiler to place the instructions from the true and false branches in an *interleaved* manner, making one instruction from the false path immediately follow one instruction from the true path, as shown in the top half of Figure 2. A null operation is placed when the true and false paths are unequal in length. Since the execution of a pair of instructions is mutually exclusive (only one of them will be executed), we select the appropriate instruction in the decode stage and pass it on to the decompressor, while the other instruction is discarded.

The lower half of Figure 2 illustrates predication through an example. A special `setpred` AX instruction precedes the sequence of interleaved code. 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 (for example, `eq`

and ne). 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. In our example, when we examine the AXThumb code, we observe that the condition in this case is eq and count is two, since there are two pairs of instructions that are conditionally executed. If eq is true, the first instruction in each pair (the add instruction) is executed; otherwise, the second instruction in each pair (the sub instruction) is executed. Therefore, after the AXThumb instructions are processed by the decode stage, the corresponding ARM instruction sequence generated consists of three instructions. The sequence contains either the add instructions or the sub instructions depending upon the eq flag.

Results

The Instruction Coalescing mechanism was implemented as part of the SimpleScalar/ARM [1] simulator using exactly one free opcode in the Thumb instruction set (ARM Architecture version ARMv5TE) to implement the AX extensions. The CACTI [9] energy model was used to compute the I-cache energy. The benchmark programs used are taken from the MediaBench [5], CommBench [6], and NetBench [11] benchmark suites. The results from our experiments are shown in Figure 3. We see that the performance gap between 32-bit code and 16-bit code has been reduced without detriment to the code size and energy saving properties of 16-bit code. The code size is almost the same as Thumb code, since most AXThumb transformations do not change the number of instructions in the program. The I-cache energy reduction in AXThumb compared to Thumb is the result of fewer fetches in AXThumb compared to Thumb. This is due to better utilization of instruction fetch queue, resulting in fewer wasted fetches. A more detailed analysis of the results can be found in [3].

With the incorporation of more AX-type coalescable instructions one can further improve the performance of 16-bit code. Thus, the design of a bridging instruction set like the AX extension, in addition to the regular 16-bit and 32-bit ISA in dual-width ISAs, can effectively bridge the performance gap between 16-bit and 32-bit code.

Conclusion

Mixed-width instruction set processors provide a unique opportunity to generate executables that have the three properties essential to the embedded computing realm: small code size, good performance, and low energy consumption. We described a profile-guided approach here that chooses instructions sizes at the granularity of functions. We have also described Instruction Coalescing, a technique that can be used to design 16-bit bridging instructions that can bridge the performance gap between 32-bit and 16-bit code. By using AXThumb instructions we support predication in Thumb state, which is currently available only in 32-bit ARM state. These techniques can be used in tandem to produce executables with the three properties essential to the embedded domain. **C**

References

1. Burger, D. and Austin, T.M. *The SimpleScalar Tool Set, Version 2.0*. Tech. Rep. 1342, Computer Sciences Dept., University of Wisconsin-Madison, 1997.
2. Intel Corporation. *The Intel Xscale Microarchitecture Technical Summary*. 2000.
3. Krishnaswamy, A. and Gupta, R. *Instruction Coalescing for 16-bit Code.* Tech. Rep. TR01-03, Department of Computer Science, The University of Arizona, Jan. 2003.
4. Krishnaswamy, A. and Gupta, R. Profile guided selection of ARM and Thumb instructions. In *Proceedings of the ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems* (June 2002), 55–64.
5. Lee, C., Potlonjak, M., and Mangione-Smith, W. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the International Symposium on Microarchitecture* (Dec. 1997).
6. Memik, G., Mangione-Smith, W., and Hu, H. NetBench: A benchmarking suite for network processors. In *Proceedings of the IEEE International Conference on Computer-Aided Design* (Nov. 2001), 39–42.
7. MIPS Technologies. *MIPS32 Architecture for Programmers Volume IV-a: The MIPS16 Application Specific Extension to the MIPS32 Architecture.* 2001.
8. Montanaro, J. et al. A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor. *IEEE Journal of Solid-State Circuits 31*, 11 (Nov. 1996), 1703–1714.
9. Reinman, G. and Jouppi, N. *An Integrated Cache Timing and Power Model.* Tech. Rep., Western Research Labs, 1999.
10. Seal, D., Ed. *ARM Architecture Reference Manual, 2d. ed.* Addison-Wesley, Reading, MA, 2000.
11. Wolf, T. and Franklin, M. CommBench: A telecommunications benchmark for network processors. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software* (Apr. 2000), 154–162.

Arvind Krishnaswamy (arvind@cs.arizona.edu) is a Ph.D. student in computer science at the University of Arizona.
Rajiv Gupta (gupta@cs.arizona.edu) is a professor of computer science at the University of Arizona.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

© 2003 ACM 0002-0782/03/0800 $5.00