Profile Guided Selection of ARM and Thumb Instructions

Arvind Krishnaswamy
Department of Computer Science
The University of Arizona
Tucson, Arizona 85721
arvind@cs.arizona.edu

Rajiv Gupta
Department of Computer Science
The University of Arizona
Tucson, Arizona 85721
gupta@cs.arizona.edu

ABSTRACT

The ARM processor core is a leading processor design for the embedded domain. In the embedded domain, both memory and energy are important concerns. For this reason the 32 bit ARM processor also supports the 16 bit Thumb instruction set. For a given program, typically the Thumb code is smaller than the ARM code. Therefore by using Thumb code the I-cache activity, and hence the energy consumed by the I-cache, can be reduced. However, the limitations of the Thumb instruction set, in comparison to the ARM instruction set, can often lead to generation of poorer quality code. Thus, while Thumb code may be smaller than ARM code, it may perform poorly and thus may not lead to overall energy savings.

We present a comparative evaluation of ARM and Thumb code to establish the above claims and present analysis of Thumb instruction set restrictions that lead to the loss of performance. We propose profile guided algorithms for generating mixed ARM and Thumb code for application programs so that the resulting code gives significant code size reductions without loss in performance. Our experiments show that this approach is successful and in fact in some cases the mixed code outperforms both ARM and Thumb 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

16/32 bit instructions, code size and speed, low power

LCTES'02–SCOPES'02, June 19-21, 2002, Berlin, Germany.

1. INTRODUCTION

In the embedded domain, applications must often execute under constraints of limited memory and they must also be energy efficient. One avenue of reducing the memory needs of an application program is through the use of code compression techniques. The ARM family of processors provides a unique opportunity for code size reduction. In addition to supporting the 32 bit ARM instruction set, these processors also support a 16 bit Thumb instruction set. By using the Thumb instruction set it is possible to obtain significant reductions in code size in comparison to the corresponding ARM code. Our experiments show that often these reductions are in the neighborhood of 30%. The MIPS-16[8] embedded processor also supports this dual instruction set feature.

As a result of the reduction in code size, the instruction cache energy expended in Thumb mode is also significantly lower in comparison to the ARM code. In our experiments a savings of up to 19% in instruction cache energy was observed. The instruction cache energy is significant percentage of total energy expended in embedded processors. In fact in a recent study it was shown that for a system consisting of a 4 issue CPU with a memory hierarchy consisting of separate L1 and L2 instruction and data caches, and a low power disk, the L1 instruction cache energy was 22% of total energy expended by the system [3]. The L1 instruction cache used in this study was a 2-way associative 32Kb cache which had a line size of 16 words. In contrast the ARM processor is a single issue processor with a 32-way instruction cache. Therefore the energy expended by the instruction cache is an even greater percentage of total energy expended.

While the use of Thumb instructions generally gives smaller code size and lower instruction cache energy, there are certain problems with using the Thumb mode. In many cases the reductions in code size are obtained at the expense of a significant increase in the number of instructions executed by the program. In our experiments this increase ranged from 9% to 41%. In fact in case of one of the benchmarks, the increase in dynamic instruction count was so high that instead of obtaining reductions in cache energy used, we observed an increase in the total amount of energy expended by the instruction cache.

From the above discussion it is clear that approaches are needed to generate mixed ARM and Thumb code that simultaneously provides compact code size, low energy, and good performance. We present a profile guided approach for generating mixed code which maximizes the use of Thumb code, in order to obtain small code size, without causing appreciable loss in performance. For those functions in which significant amounts of execution time is spent, if the Thumb code runs slower than the ARM code, we choose to either use ARM code or generate a mixture of ARM and Thumb

code. The remainder of the program is compiled into Thumb code. We present a number of different heuristics, each having a different cost associated with them, to identify functions that should be compiled into ARM code and compare their performance. We demonstrate that in general these approaches are quite effective in simultaneously reducing code size and achieving good performance. The reduction in I-cache activity also reduces the energy consumed by the I-cache. Reducing cache energy is important because for the ARM processor the cache energy can account for around 40% of total energy consumed by the processor.

The main contributions of this paper are:

- We carry out a comparative evaluation of ARM and Thumb code for a set of applications taken from the Mediabench suite. We show that typically ARM code runs faster than Thumb code but the Thumb code is of smaller size and it therefore also provides better I-cache behavior.
- The main causes of the loss in performance when running the Thumb code, in comparison to the ARM code, are identified by comparing the code generated for the above applications. The differences in the codes are traced back to the difference in the ARM and Thumb instruction sets.
- Profile guided algorithms for generating mixed code to achieve both compact code size and good performance are presented. We show that this approach is successful. The algorithms presented differ in the cost of identifying the code segments that must be compiled into ARM code. All remaining code is compiled into Thumb code.

The remainder of the paper is organized as follows. In section 2 a brief overview of the ARM processor architecture is given. Section 3 presents a comparative evaluation of ARM and Thumb code. In sections 4 and 5 we present two approaches for profile guided generation of mixed ARM and Thumb code along with their evaluation. Conclusions are given in section 6.

2. ARCHITECTURE OVERVIEW

2.1 Processor Pipeline

ARM processor core is often used as a macrocell is building application specific system chips. At the same time a number of standard CPU chips based upon the ARM core are also available [2] (e.g., ARM810, StrongARM SA-110 [4], XScale [5]). While the pipelines used by each of these CPU chips varies, they all perform in order execution. Our work is based upon the StrongARM SA-110 pipeline which consists of five stages: (i) instruction fetch; (ii) instruction decode and register read; branch target calculation and execution; (iii) Shift and ALU operation, including data transfer memory address calculation; (iv) data cache access; and (v) result write-back to register file.

2.2 Thumb Implementation

The Thumb instruction set is easily incorporated into an ARM processor with a few simple changes. The basic instruction execution core of the pipeline remains the same because it is designed to only execute ARM instructions. A Thumb instruction decompressor is added to the instruction decode stage. The decompressor is designed to translate a Thumb instruction into an equivalent ARM instruction. The addition of the decompressor in series with the instruction decoder does not increase the cycle time as the ARM decoder is quite simple and does little work during the decode cycle.

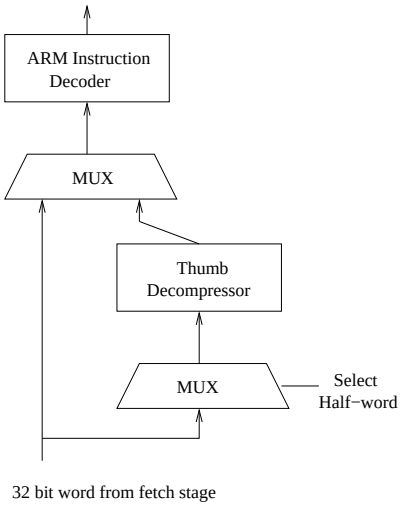

Figure 1: Thumb Implementation.

2.3 ARM vs. Thumb Instruction Sets

Some of the important differences between the two 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 one of the sources) while ARM supports 3-address format for manipulating 32 bit data. Visible registers in Thumb mode are `r0` through `r7`; only some instructions, mainly `MOVE` and `ADD` instructions, can directly address registers `r8` through `r15`. In ARM mode all sixteen registers from `r0` through `r15` are visible.

2.4 The Branch and Exchange Instruction

This instruction can be used to switch between ARM and Thumb modes. The current mode 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 mode is switched. Let us assume that the processor is in ARM mode. The `BX Rm` instruction provides the ability for the processor to switch to executing Thumb instructions as follows. When executing ARM instructions, the execution of `BX Rm` instruction can be used to begin executing Thumb instructions. `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. Similarly the `BX` instruction can be used to switch from Thumb mode to ARM mode.

3. ARM CODE VS. THUMB CODE

We started out by carrying out a study which compared the characteristics, both code size and performance, of ARM only and Thumb only versions of application programs. The purpose of this study was to first experimentally establish our claim that typically Thumb code is smaller in size while ARM code executes faster. We also wanted to gain insights into the reasons for the above behavior and how mixed code may be generated to achieve both small code size and good performance. Before we describe the results of this study, we describe the experimental set up used in this work.

3.1 Experimental Setup

Processor simulator

We started out with a port of Simplescalar [1] to ARM available from the University of Michigan. This version simulates the five stage pipeline described in the preceding section which is the Intel's SA-1 StrongARM pipeline [4] found in for example the SA-110. 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 timing of the model has been validated against a Rebel NetWinder Developer workstation [10].

We have extended the above simulator in two important ways for this research. First we modified Simplescalar to use the system call conventions followed by the `Newlib` C library instead of `glibc` which it currently uses. We made this modification because `Newlib` has been developed for use by embedded systems [6]. Second we incorporated the implementation of Thumb instruction set into Simplescalar. The original version of the simulator was built to only execute ARM code. The mechanism for switching between Thumb and ARM modes was implemented. The instruction fetch mechanism also had to be modified to appropriately deal with fetches of Thumb instructions. The semantics of the `BX` instruction was implemented to switch processor modes.

Optimizing compiler

The compiler we used in this work is the `gcc` compiler which was built to create a version that supports generation of mixed ARM and Thumb code. Specifically we use the `xscale-elf-gcc` compiler version `2.9-xscale`. Each module in the application can be compiled into either Thumb code or ARM code. The transitions between the modes at function boundaries are also taken care of by the compiler. From the above perspective, the libraries are treated as a single module, that is, either they are compiled into ARM code or completely into Thumb code. All programs were compiled at -O2 level of optimization. We did not use -O3 because at that level of optimization function inlining and loop unrolling is enabled. Clearly since the code size is an important concern for embedded systems, we did not want to enable function inlining and loop unrolling.

Representative benchmarks

The benchmarks we have used in this work are taken from the `Mediabench` [7] suite as they are representative of a class of applications important for the embedded domain. The following programs are used in this work:

- Adaptive differential pulse code modification audio coding: `adpcm` - `rawcaudio` and `rawdaudio`.
- Voice compression coder based on G.711, G.721, and G.723 standards: `g721` - `decode` and `encode`.
- A lossy image compression decoder: `jpeg` - `cjpeg` and `djpeg`.
- OpenGL graphics clone: using Mipmap quadrilateral texture mapping: `mesa` - `mipmap`, `osdemo`, & `texgen`.
- A public key encryption scheme: `pegwit` - `gen`, `encrypt`, and `decrypt`.

3.2 Performance Data

The comparison of code sizes and execution times of the ARM and Thumb versions of the programs are made in Tables 1 and 2 respectively. The size of the Thumb code is always smaller by around 30%. However, the execution times of the Thumb code exceed that of the ARM code in all benchmarks except `pegwit`. The execution time of the Thumb code exceeds that of ARM code by a small amount in some cases (e.g., around 5% for `g721`) while in other cases the Thumb execution time exceeds that of ARM execution time by a much higher amount (e.g., up to 30% for `adpcm`).

Two factors impact the relative execution times of ARM and Thumb codes: *instruction cache behavior* which is usually better for the Thumb code; and the *instruction counts* which are better (i.e., lower) for the ARM code. The data in Table 3 shows two things. First, the I-cache misses for the Thumb code are significantly lower than I-cache misses encountered by the ARM code. Second, even though the Thumb code executes greater number of instructions, the number of I-cache accesses made by the Thumb code are lower because typically each cache access fetches two useful Thumb instructions. The only exception is `adpcm` for which Thumb code performs around 20% more I-cache accesses than the ARM code. Since the I-cache behavior of the Thumb code is superior to that of ARM code, in all cases except `adpcm`, we also observed net savings in I-cache energy as shown in Table 4. These energies were computed using the cache energy models available through the `cacti` tool [12, 9].

The data in Table 5 compares the number of instructions executed by the Thumb and ARM codes. As we can see, the number of instructions executed by Thumb code are significantly higher. The increase ranges from around 9% to around 41%. 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. More importantly we can conclude that while Thumb code is smaller and typically expends lesser amount of I-cache energy, these improvements are at the cost of net performance loss as the corresponding ARM code gives lower cycle counts.

4. COARSE GRAINED GENERATION OF MIXED CODE

The basic approach that we take for generating mixed code consists of two steps. First we find the frequently executed functions once using profiling (e.g., using `gprof`). These are functions which take up more than 5% of total execution time. Second we use heuristics for choosing between ARM and Thumb codes for these frequently executed functions. For all other functions, we generate Thumb code. The above approach is based upon the observation that we should use Thumb mode whenever possible. Functions for which the use of Thumb code results in significantly lower overall performance must be compiled into ARM code. Since each function is either compiled entirely into Thumb code or entirely into ARM code, we refer to this approach as the *coarse grained* approach.

4.1 Heuristics

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. Based upon the expected performance of the two versions of the functions (ARM and Thumb) and their relative code sizes we make the final decision. We considered four different methods for making these decisions. The reasons for considering these different methods is the variation in their costs.

Table 1: Code Size Comparison.

Benchmark	ARM	Thumb	$\frac{Thumb}{ARM}$
adpcm.rawcaudio	34096	23664	.6940
adpcm.rawdaudio	34080	23652	.6940
g721.encode	40552	28456	.7017
g721.decode	40576	28448	.7011
jpeg.cjpeg	106088	72344	.6819
jpeg.djpeg	121628	83908	.6898
mesa.mipmap	535352	387132	.7231
mesa.osdemo	564632	406344	.7196
mesa.texgen	533284	385824	.6833
pegwit.gen	72320	49304	.6817
pegwit.encrypt	72320	49304	.6817
pegwit.decrypt	72320	49304	.6817

Table 2: Cycle Count Comparison.

Benchmark	ARM	Thumb	$\frac{Thumb}{ARM}$
adpcm.rawcaudio	2071274	2566574	1.2391
adpcm.rawdaudio	6942616	9055648	1.3043
g721.encode	361119677	381726744	1.0571
g721.decode	354906683	372925996	1.0571
jpeg.cjpeg	20614675	21685533	1.0519
jpeg.djpeg	5802652	7153019	1.2327
mesa.mipmap	2653507400	3175706544	1.1967
mesa.osdemo	251196387	288795297	1.1496
mesa.texgen	3762798283	4047500837	1.0757
pegwit.gen	63909156	63710588	0.9968
pegwit.encrypt	84410014	84294656	0.9986
pegwit.decrypt	64980516	63055496	0.9704

Table 3: Instruction Cache Behavior Comparison.

Benchmarks	Cache Misses		Cache Accesses		
	ARM	Thumb	ARM	Thumb	$\frac{Thumb}{ARM}$
adpcm.rawcaudio	210	173	2085582	2487321	1.1926
adpcm.rawdaudio	209	161	7142252	8656657	1.2120
g721.encode	372	282	388292504	356402511	0.9179
g721.decode	366	284	383378854	347620045	0.9067
jpeg.cjpeg	1626	1049	21607218	18529610	0.8575
jpeg.djpeg	1436	1049	5838698	4783569	0.8192
mesa.mipmap	78698	2573	2846469261	2678714821	0.9410
mesa.osdemo	124121	60889	253706375	234808518	0.9255
mesa.texgen	3649717	204271	3951507585	3610368903	0.9137
pegwit.gen	842	597	18478306	15804550	0.8553
pegwit.encrypt	1032	729	39739724	32341975	0.8138
pegwit.decrypt	947	671	18819087	16087933	0.8549

Table 4: Instruction Cache Energy (Joules).

Benchmark	ARM	Thumb	$\frac{Thumb}{ARM}$
adpcm.rawcaudio	0.18995	0.22654	1.1925
adpcm.rawdaudio	0.65047	0.78838	1.2120
g721.encode	36.1547	32.2162	0.8910
g721.decode	35.7717	31.4428	0.8789
jpeg.cjpeg	1.96793	1.68759	0.8575
jpeg.djpeg	0.53186	0.43551	0.8188
mesa.mipmap	259.208	243.952	0.9411
mesa.osdemo	23.1165	21.3897	0.9252
mesa.texgen	360.199	328.817	0.9128
pegwit.gen	1.68291	1.43938	0.8552
pegwit.enc	3.61917	2.94528	0.8138
pegwit.dec	1.71395	1.46519	0.8548

Table 5: Instruction Count Comparison.

Benchmark	ARM	Thumb	$\frac{Thumb}{ARM}$
adpcm.rawcaudio	1930162	2551796	1.3220
adpcm.rawdaudio	6380437	9024731	1.4144
g721.encode	297717704	350277214	1.1283
g721.decode	293620043	343674187	1.1334
jpeg.cjpeg	17969381	20183770	1.1232
jpeg.djpeg	4854534	6485051	1.3358
mesa.mipmap	2169393328	2748271643	1.2668
mesa.osdemo	192999016	240092504	1.2440
mesa.texgen	2918265376	3470919050	1.1893
pegwit.gen	15758948	17233370	1.0935
pegwit.encrypt	33906995	36880766	1.0877
pegwit.decrypt	16045351	17529285	1.0935

Heuristic I

This method is the most precise. The ARM and Thumb versions of the program are executed on the simulator and the cycle counts for each relevant function are measured. If, for a given function, the cycle count for the Thumb version is lower than the ARM version, then we choose to use the Thumb version for that function; otherwise we use the ARM version. The advantage of using cycle counts is that it takes into account not only the number of instructions executed by the two code versions, but also the cache behaviors of the two code versions.

While the above approach is quite precise because it uses the cycle counts to determine which of the two versions is superior, ARM or the Thumb, it is also expensive since the executions of the ARM and Thumb versions of the code must be simulated to obtain the cycle counts. In fact this is the most expensive method that we propose.

Heuristic II

In this method, instead of using cycle counts for functions, we simply use instruction counts. In other words we use Thumb code for a function if that gives a lower instruction count than the corresponding ARM version; otherwise we use the ARM version. The instruction counts are less precise because they do not account for I-cache behavior. Recall that Thumb code usually experiences fewer cache misses.

While this second approach is less precise, it is also less expensive. We no longer need to simulate the executions of ARM and Thumb versions of a program. Instead we need to simply collect profiles in form of basic block counts by executing the ARM and Thumb versions of the code.

Heuristic III

This heuristic simply uses the relative sizes of ARM and Thumb code to decide upon which version to use. Note that if the ARM version were perfectly compressed during generation of the Thumb version, then we can expect the size of the Thumb version to be half the size of the ARM version as the instruction size is halved when we go from 32 bit to 16 bit instructions. However, as we know from the data presented earlier, the Thumb code is typically only 30% smaller than the ARM code. Less compression implies that Thumb code contains extra instructions which may be executed at runtime. Therefore we set a threshold value `T` such that if the Thumb code for a function is more than `T`% smaller than the corresponding ARM code, then we use the Thumb code; otherwise we use the ARM code for the function.

While this approach requires generation of ARM and Thumb versions of the code for each function, it does not require their execution. Therefore this method is very inexpensive. Of course this is the most approximate method among the three heuristics we have presented so far.

Heuristic IV

In this final method we use a combination of instruction counts and relative code sizes to make the decisions. In particular we use the Thumb code if one of the following conditions hold: (a) the Thumb instruction count is lower than the ARM instruction count; or (b) the Thumb instruction count is higher by no more than `T1`% and the Thumb code size is smaller by at least `T2`%.

The idea behind this heuristic is that if the Thumb instruction count for a function is slightly higher than the ARM instruction count, it still may be fine to use Thumb code if it is sufficiently smaller than the ARM code as the smaller size may lead to fewer instruction cache accesses and misses for the Thumb code. Therefore the net effect may be that the cycle count of Thumb code may not be higher than the cycle count for the ARM code.

The cost of this method is no more than heuristic II as we only need the code sizes of the ARM and Thumb versions and basic block counts collected through profiling.

4.2 Implementation

In our implementation, the above heuristics were applied at module level and not at individual function level. That is, all functions in a module are either compiled into Thumb code or all are compiled into ARM mode. This approach works well because if closely related functions are compiled into different modes, optimizations across function boundaries are disabled and their is a loss in performance as a result. From the above perspective, the libraries used by a program are treated as a single module, that is, either we link the program with a version of the libraries that are completely compiled to Thumb code or to a version that is completely compiled into ARM code.

4.3 Results

The results of our experiments are shown in Tables 6-8. As we can see for the results, the *Mixed* code size is significantly smaller than the ARM code size. In fact the Mixed code size is only slightly bigger than the Thumb code size. We also observe that the Mixed code gives instruction cache energy savings over the ARM code. Moreover the energy savings are comparable to those obtained over Thumb code (in some cases they are bit higher and others lower and for some they are unchanged). Finally the cycle count of the Mixed code is very close to the cycle count of the ARM code. In fact in some cases it is even slightly smaller.

In summary we can say that the size of the Mixed code is close to that of Thumb code and its performance is close to that of ARM code. Therefore this profile guided approach gives best of both worlds – smaller code size and good performance.

Now let us compare the results of heuristic I with that of the other three heuristics (II, III, and IV) which are less expensive but not as precise as heuristic I. For heuristic III we set the threshold T to 35% and for heuristic IV we set the thresholds T1 to 3% and T2 to 40%. For all heuristics, when we encountered a module with functions that needed to be compiled differently we chose to compile the whole module as ARM code rather than splitting the module and compiling them differently. In addition to making the heuristics more general and not specializing them for these cases, it also enables better performance in some cases as the compiler now has the opportunity to carry out interprocedural optimizations of static functions in the same module. It is also possible that the performance decreases due to a large number of functions in a module are compiled in ARM rather than Thumb worsening the cache performance.

By comparing the results we see identical results for all heuristics for `adpcm`, `mesa`, `g721` and `pegwit.gen`. Although the results are the same, identical decisions were made only for `adpcm` and `mesa`.

In the other two cases the module splitting caveat described above caused the change in the decisions made by the heuristics resulting in identical results. For the other cases we notice that as expected heuristic II is either as good as heuristic I or slightly worse. Using heuristic III we get better performance than heuristic II for `jpeg.cjpeg`. This is because certain functions which were compiled as Thumb using heuristic II have been compiled as ARM using heuristic III. The lower I-cache performance and larger code

Table 6: Code Size Comparison.

Benchmark	$\frac{Thumb}{ARM}$	$\frac{Mixed}{ARM}$			
		I	II	III	IV
adpcm.rawcaudio	0.694	0.695	0.695	0.695	0.695
adpcm.rawdaudio	0.694	0.695	0.695	0.695	0.695
g721.encode	0.701	0.719	0.719	0.719	0.719
g721.decode	0.701	0.719	0.719	0.719	0.719
jpeg.cjpeg	0.681	0.715	0.696	0.730	0.696
jpeg.djpeg	0.689	0.711	0.711	0.700	0.711
mesa.mipmap	0.723	0.772	0.772	0.772	0.772
mesa.osdemo	0.719	0.766	0.766	0.766	0.766
mesa.texgen	0.723	0.771	0.771	0.771	0.771
pegwit.gen	0.681	0.715	0.715	0.715	0.715
pegwit.encrypt	0.681	0.822	0.822	0.715	0.681
pegwit.decrypt	0.681	0.822	0.822	0.715	0.681

Table 7: Instruction Cache Energy Comparison.

Benchmark	$\frac{Thumb}{ARM}$	$\frac{Mixed}{ARM}$			
		I	II	III	IV
adpcm.rawcaudio	1.926	0.999	0.999	0.999	0.999
adpcm.rawdaudio	1.212	0.999	0.999	0.999	0.999
g721.encode	0.920	0.992	0.992	0.992	0.992
g721.decode	0.907	0.983	0.983	0.983	0.983
jpeg.cjpeg	0.858	0.903	0.856	0.923	0.856
jpeg.djpeg	0.819	0.866	0.866	0.815	0.866
mesa.mipmap	0.941	0.986	0.986	0.986	0.986
mesa.osdemo	0.926	0.980	0.980	0.980	0.980
mesa.texgen	0.914	0.982	0.982	0.982	0.982
pegwit.gen	0.855	1.130	1.130	1.130	1.130
pegwit.encrypt	0.814	1.283	1.283	1.283	0.814
pegwit.decrypt	0.855	1.325	1.325	1.325	0.855

Table 8: Cycle Count Comparison.

Benchmark	$\frac{Thumb}{ARM}$	$\frac{Mixed}{ARM}$			
		I	II	III	IV
adpcm.rawcaudio	1.239	0.999	0.999	0.999	0.999
adpcm.rawdaudio	1.304	0.999	0.999	0.999	0.999
g721.encode	1.057	1.033	1.033	1.033	1.033
g721.decode	1.057	1.039	1.039	1.039	1.039
jpeg.cjpeg	1.051	1.024	1.047	1.033	1.047
jpeg.djpeg	1.232	1.145	1.145	1.234	1.145
mesa.mipmap	1.196	1.017	1.017	1.017	1.017
mesa.osdemo	1.149	1.015	1.015	1.015	1.015
mesa.texgen	1.075	1.004	1.004	1.004	1.004
pegwit.gen	0.996	0.988	0.988	0.988	0.988
pegwit.encrypt	0.998	1.193	1.193	1.193	0.998
pegwit.decrypt	0.970	1.088	1.088	1.088	0.970

size corroborate this behavior. The inverse of this situation occurs in `jpeg.djpeg`.

The results for heuristic IV are quite close to heuristic I and surprisingly better in the case of `pegwit.encrypt` and `pegwit.decrypt`. The cycle counts for these two benchmarks are the best when the whole module is compiled as Thumb. Heuristics I - III choose to compile certain functions as ARM and this results in poorer performance. This is again attributed to the module splitting caveat mentioned above. Heuristic IV, on the other hand, chooses to compile all modules in `pegwit.encrypt` and `pegwit.decrypt` as Thumb. Therefore in summary, as expected, heuristic I and IV perform better than heuristics II and III. However, although we expected that heuristic I would always perform better than heuristic IV, in the case of `pegwit.encrypt` and `pegwit.decrypt` heuristic IV gives the best performance. In any case, we can conclude that this coarse grained profile guided approach is quite effective in generating mixed code.

5. FINE GRAINED GENERATION OF MIXED CODE

In the preceding approach each function was either compiled completely into ARM code or completely into Thumb code. The next question we wanted to answer was whether a finer grained approach, in which a single function can consist of a mixture of ARM and Thumb instructions, would result in better overall results. To develop a method for making decisions at finer grained level, we begin by analyzing the ARM and Thumb codes generated for the benchmarks in the preceding section.

5.1 Analysis of Instruction Counts

We know that for some functions the Thumb version executes far greater number of instructions. We examined the dynamic instruction counts of major categories of instructions to see how each of the categories are impacted by the use of Thumb instructions. The additional instructions executed in Thumb mode were distributed among four broad types: Branches, ALU operations, register to register MOVES, and Load/Store instructions. The changes in the dynamic instruction counts are shown in Table. 9 where a positive value represents an increase and a negative value a decrease. The percentage changes were computed as follows:

$$\frac{ThumbCountTypeX - ARMcountTypeX}{TotalARMCount}.$$

There is an increase in branch instructions because Thumb does not support predication while ARM does. The large increase in ALU instructions is due to a number of reasons. For example the difficulty of supplying large immediate operands in Thumb mode result in extra instructions. The load/store instruction increase because there are fewer registers directly available to ALU instructions in Thumb mode and these are assigned to variables for shorter periods of time.

Surprisingly we found that the number of MOVE instructions is often lower for the Thumb code. However, upon closer examination we found that this is actually not the case. When shift operations are needed, the Thumb code generates explicit shift instructions which were counted as ALU instructions. However, in ARM mode, MOVE instructions support a shift amount field which caused shift instructions to be generated as MOVE instructions.

Next we looked for patterns of equivalent instruction sequences in Thumb and ARM versions of a function that clearly account for significant changes in dynamic instruction counts. Our approach for generating mixed code for a given function is based upon these patterns. Next we describe our approach.

Table 9: Extra Instructions Executed in Thumb Mode.

Benchmark	Branch	ALU	MOVES	Ld/St	Total
adpcm.rawcaudio	21.08%	-0.08%	9.06%	2.87%	32.93%
adpcm.rawdaudio	20.78%	14.84%	11.60%	-5.78%	41.44%
g721.encode	5.36%	17.39%	-6.07%	0.96%	17.64%
g721.decode	4.90%	16.20%	-4.56%	0.50%	17.04%
jpeg.cjpeg	5.53%	9.76%	5.37%	-8.34%	12.32%
jpeg.djpeg	1.02%	20.88%	9.62%	1.02%	32.54%
mesa.mipmap	7.57%	24.03%	-15.30%	10.38%	26.68%
mesa.osdemo	7.53%	21.77%	-13.93%	9.01%	24.38%
mesa.texgen	11.87%	19.66%	-16.89%	4.29%	18.93%
pegwit.gen	3.83%	33.62%	-24.43%	-3.67%	9.35%
pegwit.encrypt	3.39%	30.07%	-19.78%	-2.35%	11.33%
pegwit.decrypt	3.81%	33.42%	-24.27%	-3.71%	9.25%

5.2 Our Approach

Now knowing that the increase in overall Thumb instruction count is due to several of the above instruction types, we set out to find frequently occurring patterns in Thumb code that account for significant amounts of increase in the overall Thumb instruction counts. The identification of these patterns forms the basis of our fine grained approach for generating mixed code. Before we describe these patterns in detail, we present our overall approach for exploiting these patterns in generating mixed code for a given function.

We begin with the coarse grained mixed code generated by Heuristic IV described in the preceding section. Each function that is compiled into ARM is a candidate for fine grained mixed code generation. To generate code for the function, we first generate the Thumb code for the entire function. Then we identify patterns of Thumb instructions that are better executed using ARM instructions. We replace these patterns with equivalent ARM code. At the transition points we introduce a `BX` instruction. At entry point of the ARM sequence a `BX` instruction switches the processor from Thumb mode to ARM mode and the reverse happens at the exit of the ARM sequence.

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

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

Note that in the fine grained approach the module splitting caveat is no longer relevant. We begin with Thumb code for the entire program and selectively replace patterns of Thumb code by ARM code in selected functions.

5.3 Patterns

Some specific patterns that we found by comparing Thumb and ARM codes generated for the benchmarks used are described next. These patterns more specifically point to causes of increased instruction counts for Thumb code. We categorize these patterns according to the type of instructions whose counts they impact the most.

ALU Instructions

There are a couple of different patterns that we found to occur frequently in our benchmarks that resulted in an increase in the number of ALU instructions executed in Thumb mode.

The first pattern arises due to a lack of an ability to specify negative offsets in Thumb mode requires extra ALU instructions to be used. The example shown below, which is taken from versions of the ARM and Thumb codes of a function in `adpcm_coder`, illustrates this point. 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.

ARM
`str rs, [rbase - offset]`

Thumb
`mov r1, offset`
`neg r1`
`add r1, rbase`
`str rs, [r1,#0]`

Another commonly occurring pattern is as follows. In ARM code shifts can be done as part of ALU operations while they have to be done explicitly using a separate shift instruction in Thumb. This pattern is illustrated by the example that follows.

ARM
`sub r5, r3, lsl #2`

Thumb
`lsl r4, r3, #2`
`sub r5, r4`

Branch Instructions

Again there are a couple of frequently occurring patterns that belong to this category. The first reason for more branches in Thumb code is that unlike ARM mode full predication is not supported in Thumb mode. The example below illustrates this using a code fragment taken from function `emit_eobrun` from `cjpeg`. The

ldmeqia is a predicated load multiple in ARM code. Note that the last register is the pc (program counter) as this code fragment actually implements a return from a function. In Thumb mode explicit branching has been introduced. The pop instruction performs multiple load into registers.

ARM
cmp r3, #0 ldmeqia sp!, {r4, r5, r6, r7, pc}
Thumb
f352: cmp r3, #0 f354: bne f358 <emit_eobrun+0x10> f356: b f4a2 <emit_eobrun+0x156> ... f4a2: pop {r4, r5, r6, r7, pc}

Another pattern that shows use of more branches in Thumb code is as follows. In the ARM mode, we can return from a function by simply moving the contents of the link register to the program counter as shown below. In contrast, in the Thumb mode the BX instruction may be used. When this is the case, we cannot simply execute BX using the link register because it is a high register. Hence the typical sequence shown below, which requires additional instructions, is used.

ARM
mov pc, lr
Thumb
mov lowreg, LR bl <call_via_lowreg> ... <call_via_lowreg>: bx lowreg nop

MOVE Instructions

Some extra move instructions are introduced in Thumb mode during the saving and restoring of registers at function boundaries. Because the high registers (r8 through r15) can be accessed only by mov, cmp and add instructions in Thumb mode, saving of registers by the callee has to involve the moving of high registers to low registers before they can be saved. The following code fragment taken from function rgb_gray_convert illustrates this point.

ARM
stmdb sp!, {r4, r5, r6, r7, r8, r9, r10, r11,lr}
Thumb

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

Since higher order registers are not accessible directly by many instructions in the Thumb mode, extra moves are required to first move the data from a higher order to a lower order register. We illustrate this pattern through an example which shows how the base index for loads when using based indexing addressing for loads is affected. In the Thumb mode, the base is saved in one of the higher order registers. Each time a load is needed the contents are moved to a lower register and the ldr instruction is executed using this low register.

ARM
ldr reg, [reg, #offset]
Thumb
mov lowreg, highreg ldr reg, [lowreg, #offset]

5.4 Results

We applied the above fine grained approach to the benchmarks and compared its performance with that of coarse grained heuristic IV. Our implementation at this point incorporates two most frequently found patterns: the pattern involving additional branches in Thumb code due to lack of full predication and the pattern involving extra move instructions at function boundaries for saving and restoring high order registers.

The results of the experiments are shown in Table 10. For benchmarks g721 and jpeg we observe that the code size is reduced by using fine grained approach over heuristic IV because greater portions of the program are compiled into Thumb code by the fine grained approach. As a result the I-cache energy consumption is also reduced. However, instead of seeing lower cycle counts, we see a small increase. This is because the patterns of Thumb code being replaced by ARM code are small and therefore the savings achieved by using ARM instructions are often more than offset by two additional BX instructions that must be introduced.

Table 10: Fine Grained vs. Heuristic IV.

Benchmark	$\frac{Fine-Grained}{HeuristicIV}$		
	Code Size	Cycle Counts	I-Cache Energy
adpcm.rawcaudio	1.0033	1.4812	1.4421
adpcm.rawdaudio	1.0033	1.6882	1.6189
g721.encode	0.9788	1.0588	0.9424
g721.decode	0.9779	1.0516	0.9383
jpeg.cjpeg	0.9895	1.0084	0.9958
jpeg.djpeg	0.9805	1.0773	0.9419
mesa.mipmap	1	1	1
mesa.osdemo	1	1	1
mesa.texgen	1	1	1
pegwit.gen	0.9613	1.0095	0.7526
pegwit.encrypt	1.0081	1.0189	1.0140
pegwit.decrypt	1.0081	1.0049	0.9940

For the adpcm benchmark the fine grained approach performs rather poorly. This is because in this benchmarks the patterns being considered occurred very frequently and in fact after replacing the patterns of Thumb code by equivalent ARM code surrounded by BX instructions actiually increase the code size, cycle counts, and I-cache energy.

There is no change observed for the mesa benchmark because in this case heuristic IV decided to compile all application functions into Thumb code as most the time is spend in library code. Since the application functions do not account for a significant portion of the execution time, pattern replacement is not applied to these functions.

The behavior of pegwit.gen is along the same lines as g721 and jpeg. However, the same is not true for pegwit.encrypt and pegwit.decrypt. For these two programs heuristic IV chooses to compile all as application code into

Thumb due to the module splitting caveat. But for fine grained we can replace patterns without carrying out module splitting and without effecting interprocedural optimization. So inspite of heuristic IV choosing to generate all Thumb code, we take those functions which were not compiled into ARM because of module splitting and do fine grained pattern replacement in them. The result of pattern replacement is a very slight increase in code size and cycle count for both `encrypt` and `decrypt`. For `decrypt` the I-cache energy is slightly reduced while for `encrypt` it is slightly increased..

In summary we can say that surprisingly the application of fine grained pattern replacement does not yield additional improvements over coarse grained strategy. While smaller code size and lower cache energy can be achieved, the cycle counts are increased. This is because the cost of using two `BX` instructions per pattern is too high and therefore the expected improvements from using ARM code instead of Thumb code are wiped out. In fact very often the patterns we replace belong to frequently executed loops. The `BX` instructions introduced by the fine grained approach are also therefore introduced inside these loops. To avoid this problem one approach could be to translate larger code segments into ARM code so that the `BX` instructions appear outside loops. However, this approach cannot be effectively applied as a postpass because typically it requires significant changes to the program's register allocation. In contrast the coarse grained approach would place the `BX` instructions at function boundaries which may be executed much less frequently. Therefore the coarse grained approach is much more effective.

6. CONCLUSIONS

Our comparison of ARM and Thumb executables shows that ARM code gives better performance at the cost of larger code size and higher I-cache energy, while Thumb code results in smaller code size and lower I-cache energy at the cost of lower performance. Our approach for generating mixed ARM and Thumb code simultaneously delivers high performance, small code size and low I-cache energy.

We presented coarse grained heuristics with varying costs to make the decisions between using ARM and Thumb code at the module level. We also presented a fine grained method which has the ability of generating mixed ARM and Thumb code for a single function. Our results show that using the coarse grained heuristics presented it is indeed possible to simultaneously achieve good performance, small code size, and low instruction cache energy. Surprisingly in some cases the mixed code performs even better than ARM version of the program. The fine grained heuristic does not give any significant additional improvement because the cost of switching mode using `BX` instruction is too high.

7. ACKNOWLEDGEMENTS

This work is supported by DARPA award F29601-00-1-0183 and National Science Foundation grants CCR-0208756, CCR-0105535, CCR-0096122, and EIA-9806525 to the University of Arizona.

8. 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] S. Gurumurthi, A. Sivasubramaniam, M.J. Irwin, N. Vijaykrishnan, and M. Kandemir, "Using Complete Machine Simulation for Software Power Estimation: The SoftWatt Approach," *The Eight International Symposium on High Performance Computer Architecture* (HPCA-8), Feb. 2002.

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

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

[6] J. Johnston, Maintainer, Newlib - http://sources.redhat.com/newlib/.

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

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

[10] rebel.com, Netwinder Family, http://www.rebel.com/netwinder.

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

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