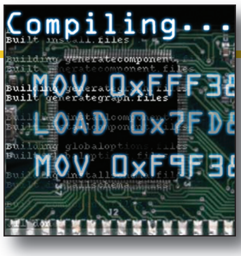

The Use of Compiler Optimizations for Embedded Systems Software

by Joe Bungo

Abstract

Optimizing embedded applications using a compiler can generally be broken down into two major categories: hand-optimizing code to take advantage of a particular processor's compiler and applying built-in optimization options to proven and well-polished code. The former is well documented for different processors, but little has been done to find generalized methods for optimal sets of compiler options based on common goal criteria such as application code size, execution speed, power consumption, and build time. This article discusses the fundamental differences between these two general categories of optimizations using the compiler. Examples of common, built-in compiler options are presented using a simulated ARM processor and C compiler, along with a simple methodology that can be applied to any embedded compiler for finding an optimal set of compiler options.

Introduction

Even with the advent of the first general-purpose computers in the 1940s, a need arose for machines designed to perform a few dedicated tasks in real-time. This need gave birth to the world's first embedded systems. In 1961, Charles Stark Draper developed The Apollo Guidance Computer at the MIT Instrumentation Lab, which generally is recognized as the first modern embedded computer system [6]. Today's definition of an embedded system would include the use of a microprocessor, which became commercially feasible around 1971 [4], allowing smaller computers to aid in making a phone call, performing surgery, or playing a game.

As embedded processor architectures have become more complicated, programmers have become more dependent on the compiler's knowledge of the processor's the instruction sets, pipelines, and complex memory systems. It is a common misconception that faster, more complex processors diminish the need for better compilers. Compiler technology must necessarily advance to take advantage of new processor features. Using a good compiler optimally could not only make code smaller and faster, but also bring financial gains during development in three basic ways:

1. **Memory Usage**
   By decreasing code size, less memory ultimately will be needed by the system. Although memory cost has gotten significantly cheaper over time, it still remains one of the most expensive components of an embedded system.

2. **Processor Selection**
   Increasing the performance of the software enables engineers to use slower, more cost-efficient processors.

3. **Time to Market**
   As compilers become more powerful, time-consuming hand optimization has become less important. Also, it has become essential to write readable, modularly structured, and maintainable code for the purpose of portability and reuse.

Let us first explore the differences between compiling (and writing) hand-optimized, high-level code for a specific architecture and using the built-in compiler optimizations that can be applied to arbitrary pieces of code. While this article touches on the differences between these two general classes of optimization techniques, it focuses mainly on the latter. Built-in compiler optimizations include one or more of the following: using options at the command-line and front-end of the tools; compiler-recognized keywords used at the source level (for example, using `__packed` might tell a compiler to reorder some declared global data in memory to eliminate padding) and automatic optimizations that the compiler will always perform. The options (also known as switches) can be applied to any piece of code, whereas the targeted, hand-optimization techniques only apply to certain aspects of code. Of course, the compiler output could vary depending on the code and switch settings, but it is possible to get the same compiler output from basic pieces of code using different architectural- and processor-specific compiler switches. Additionally, a methodology for using and selecting these built-in switches will be presented. Think of this article as more of a guide to optimizing polished source code using only compiler options (which are one step of an entire optimization process) rather than a guide to writing optimized high-level code for a specific processor.

First, it is worth having a brief look at the idea of writing code to run on a specific processor or architecture. This should not be confused with using built-in architectural- and processor-specific optimization switches that are shown later.

Hand Optimizing High-Level, Targeted Code

One simple example of writing ARM-targeted C can be found in coding loops. In C it is common practice to implement a simple for-loop incrementing a counter as:

```
int countUp() {
  unsigned int i;
  int sum = 0;
  for (i=0; i<10; i++)
    sum += 1;
  return sum; }
```

Simulating an ARM7-based NXP microcontroller [9] running at 12 MHz (the target used in this article unless otherwise specified) with a proprietary ARM C compiler and compiling at the highest optimization level (–O3, which will be discussed in more detail later), the compiler outputs the following ARM assembly code:

```
           MOV       r0, #0x00000000
           MOV       r1, r0
loop       ADD       r1, r1, #0x00000001
           CMP       r1, #0x0000000A
           ADD       r0, r0, #0x00000001
           BCC       loop
```

The compiler uses `r0` as the sum and `r1` as the counter `i`, incrementing both the counter and sum, comparing the counter to ten each time around the loop. The reasons these particular registers are used can be found in ARM's procedure call standard (called the AAPCS [1]). The `CMP` instruction sets the condition code flags based on `r1` minus ten and the `BCC` instruction indicates a branch back to the `loop` label if the carry flag is clear in the condition code register.

Profiling the code in the simulator shows that `countUp` takes 1.050 microseconds to execute. However, the compiler can do better than this by using conditional execution and flags. As a general rule, loops running on ARM processors should always be written so that the counter decrements down to zero:

```
int countDown() {
  unsigned int i;
  int sum = 0;
  for (i=10; i!=0; i--)
    sum += 1;
  return sum; }
```

With the same target and compiler settings, the compiler produced the following output:

```
           MOV       r0, #0x00000000
           MOV       r1, #0x0000000A
loop       SUBS      r1, r1, #0x00000001
           ADD       r0, r0, #0x00000001
           BNE       loop
```

The compiler initializes the loop counter to ten and decrements the counter down to zero, setting the condition code flags with each subtract (indicated by using `SUBS` versus `SUB`). The code falls out of the loop once the Z flag in the condition code register is set (meaning a zero result was produced by the `SUBS` instruction). This is one example of how the ARM processor can significantly reduce code size and execution speed via intelligent conditional execution and flag usage. Profiling the code in the simulator shows that `countDown` takes .883 microseconds to execute. The code is smaller and faster because there is no need to make a comparison (using the `CMP` instruction) each iteration of the loop. Although targeted at the ARM architecture, mostly because of its ability to conditionally execute instructions, this is an instance of a source-level, hand optimization that could work for other architectures.

Another source-level optimization that is even more specific to the ARM architecture deals with the way in which parameters are passed. Part of the procedure call standard used by the compiler states that no more than four parameters can be passed to a function by way of registers, which is a faster method than passing through stack memory. Passing more than four parameters to a function will result in the compiler spilling the fifth, sixth, seventh, etc., parameters onto stack memory. Therefore, programmers should try to pass no more than four parameters to functions if possible. If not, the most frequently used parameters should be passed before the others for the same reasons.

There are other architectures that have similar standards, i.e., MIPS, but this type of optimization is more architecture-specific than, for example, a compiler's ability to inline function calls [8]. This type of optimization lets a compiler expand a function call into the actual function body, typically improving the execution time of the application by eliminating function call overhead. However, this usually comes at the expense of code size. Inline expansion can result in further optimizations, such as the function no longer requiring procedure call restrictions. In addition to being a very common compiler optimization (even across other types of systems such as supercomputers), most developers do not write high-level code with compiler inlining in mind. It is usually left completely up to the compiler to do when activated.

Levels of Generalized Optimization

Most compilers can use a set of general optimizations that usually depend on the stage of the application's development as well as the programmer's debugging needs. In general, assembler output that has been highly optimized is more difficult to understand and, consequently, to debug. The levels of general optimizations performed by the ARM C compiler are summarized in Table 1.

Most of the optimizations detailed in this article would be performed based on the level of generalized optimization selected. There are, however, some specific compiler switches that can override options for a particular type of optimization. In the case of the ARM C compiler, there must always be a level of optimization used, whether it be the default or specified by the developer.

`-O0`	The lowest level of optimization. Simple optimizations are performed so not to impair the debug view. This switch gives the best possible debug view.
`-O1`	A restricted level of optimization giving a satisfactory debug view and good code density.
`-O2`	A high level of optimization which might give a less satisfactory debug view. This is the default option.
`-O3`	The highest, most aggressive level of optimization. The optimizations performed depend on the whether the `-Ospace` or `-Otime` options are enabled. This typically leaves a poor debug view.

Table 1: General levels of optimization options for ARM C compiler.

The Fundamental Compiler Optimization Choice: Code Size vs. Execution Speed

Many compilers allow developers to specify whether they want their code to be optimized more toward performance or code size. Speed used to be the most crucial, however, that has started to change as embedded systems get smaller and have more limited resources. Many modern embedded systems do not boot from a hard disk, but instead, use limited amounts of non-volatile memory such as Flash and ROM. Also, many of these systems do not utilize virtual memory which limits the amount of RAM available.

Most compilers have different switches to signify optimization for size or speed, depending on the goals of the application. Typically it is possible to apply different switches to different portions of code. The ARM C compiler uses the following:

`-Ospace`
This switch, enabled by default, tells the compiler to apply optimizations for reducing image size, possibly at the expense of speed.

`-Otime`
This switch tells the compiler to apply optimizations for execution speed, possibly at the expense of code size.

As a trivial example, suppose we have a function which calculates a 16-bit **`checksum`** of a data packet containing at least ten 16-bit values and then adds five to it. A **`main`** function defines the packet and calls the **`checksum`** function:

```
short checksum(short *data, int n) {
  unsigned int i;
  int sum = 0;
  for (i = 10; (i+n) != 0; i--)
    sum += *(data++);
  return (sum+5); }

int main() {
  short a[15] =
    {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
  checksum(a,5);
  return 0; }
```

Compiling with the `-Otime` option produces the following ARM assembly for the checksum function:

```
checksum   ADDS      r12, r1, #0x0000000A
           MOV       r3, #0x00000000
           MOV       r2, #0x0000000A
           BEQ       done
loop       LDRSH     r12, [r0], #0x02
           SUB       r2, r2, #0x00000001
           ADD       r3, r3, r12
           ADDS      r12, r2, r1
           BNE       loop
done       ADD       r0, r3, #0x00000005
           MOV       r0, r0, LSL #16
           MOV       r0, r0, ASR #16
           BX        r14
```

It should be noted that this output was produced by also applying the `-O3` and **`no_inline`** options. The **`no_inline`** option prevents the compiler from inlining any functions and is only applied here to illustrate concise differences between the `-Otime` and `-Ospace` options. Otherwise, the **`checksum`** function would have been expanded inside of main.

Comparing this to applying the `-Ospace` switch, again also using the `-O3` option:

```
checksum   MOV       r3, #0x00000000
           MOV       r2, #0x0000000A
loop       ADDS      r12, r2, r1
           LDRNESH   r12, [r0], #0x02
           SUBNE     r2, r2, #0x00000001
           ADDNE     r3, r3, r12
           BNE       loop
           ADD       r0, r3, #0x00000005
           MOV       r0, r0, LSL #16
           MOV       r0, r0, ASR #16
           BX        r14
```

In the `-Otime` output the compiler duplicates the loop condition test once outside of the loop before the actual loop begins. This results in one less loop iteration compared to the `-Ospace` output, which speeds up execution of the algorithm but requires one more instruction. See Table 2 for the code size and execution speed comparisons based on simulations with the tools. Since this is an instruction level simulator of a microcontroller running at a fixed frequency, the execution times do not vary over multiple runs depending on host PC factors such as memory usage and background processes or threads.

The **`LDRNESH`** instruction indicates signed halfword (16-bit) loads from memory if the Z flag is clear in the condition code register (meaning a non-zero result was produced by the **`ADDS`** instruction) using post-increment addressing. The **`BX`** instruction is a specific type of branch instruction that the compiler will use to return from functions using a return address that was previously stored in the link register (**`r14`**). The last thing to note is the shifts **`LSL`** and **`ASR`** (Logical Shift Left and Arithmetic Shift Right) at the end of both functions. Since we are returning a piece of short data, the compiler must ensure that the return value only occupies the bottom 16 bits of the register since all ARM registers are 32 bits. For this reason, 32-bit integer data types should be used for local variables when possible. This will eliminate any shifting and masking instructions that ensure data only occupies the appropriate bits of a register.

In the case of the ARM C compiler, `-Otime` or `-Ospace` will always be applied, whether it be the default or specified by the developer.

	Execution Time (microseconds)	Program Size* (bytes)
Compiled with `-Otime`, `-O3`	2.383	150
Compiled with `-Ospace`, `-O3`	2.467	134

*Program size reflects object file that includes **`main`** and all associated data, but not startup code.

Table 2: `Checksum` simulation comparisons.

Architectural- and Processor-Specific Optimizations

As processor architectures advance, new instructions and enhancements are introduced. In many instances the compiler must be made aware of the target processor and architecture in order to take advantage of the new features. Some examples include having the compiler use new DSP instructions to speed up some mathematical library routines, as well as allowing a processor to perform unaligned memory accesses that would perhaps cause faults on a different processor.

Another reason you might specify this information to the compiler is to allow it to appropriately schedule instructions to make use of a particular processor's pipeline [10]. Instruction scheduling can be used, for example, to avoid interlocks or maybe to take advantage of a dual issue pipeline. A dual issue pipeline has the ability to pass two instructions from a decode stage to an execute stage in the same cycle [7, 12].

Consider the following C code:

```
void schedule(int *p, int *q, int z) {
  int x = *p;
  int y = *q;
  x = x * x;
  z = z + 1; }
```

Unless specified otherwise, the ARM C compiler will always schedule instructions for an ARM9 pipeline (which has no effect on code that will run on earlier cores), resulting in the following assembly output:

```
schedule    LDR       r0, [r0]
            LDR       r1, [r1]
            MUL       r0, r0, r0
            ADD       r2, r2, #1
```

However, this code would not take advantage of a processor's dual-issue capability, which the ARM Cortex-R4 core (which implements ARM architecture version 7-R) and Cortex-A8 core (which implements ARM architecture version 7-A) both have. By applying the `--cpu=7-R` or `--cpu=7-A` compiler switch to specify an architecture that can perform dual-issuing [12], the output would look like:

```
schedule    LDR       r0, [r0]
            ADD       r2, r2, #1
            LDR       r1, [r1]
            MUL       r0, r0, r0
```

Since memory access and data processing instructions have separate back-end pipelines on these cores (see Figure 1), the initial `LDR` and `ADD` instructions can be executed simultaneously, allowing for increased performance.

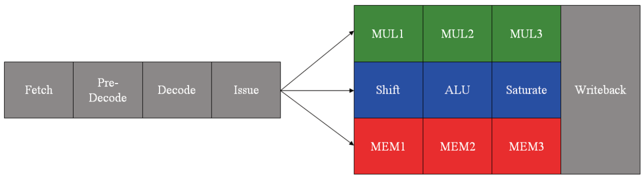

Figure 1: The Cortex R4's Dual-issue Pipeline.

Multiply operations also have their own backend pipeline. The Pre-Decode stage allows for branch prediction and some necessary instruction formatting.

Compressed Instruction Sets

Most ARM cores have the option to execute a compressed version of the standard instruction set. The 16-bit Thumb [5] instruction set is a compressed subset of the standard 32-bit ARM instruction set and can provide not only better code density, but also better performance on 16-bit memory systems. These additional instruction sets are relevant because they are usually generated by the compiler using a switch as opposed to hand-coding them (which can prove difficult to do). Also, it may take multiple 16-bit Thumb instructions to accomplish what one 32-bit ARM instruction can do—yet another reason to leave the decision about which instruction set to use up to the compiler.

Let us revisit the `checksum` example. If we recompile this example with the `-O3`, `-Ospace`, and now the `--thumb` option, we get the following Thumb assembly output:

```
checksum    PUSH      {r4}
            MOV       r2, #0x0A
            B         start
loop        MOV       r4, #0x00
            LDRSH     r4, [r0, r4]
            ADD       r0, r0, #2
            SUB       r2, r2, #1
            ADD       r3, r4, r3
start       ADD       r4, r2, r1
            BNE       loop
            ADD       r3, r3, #5
            POP       {r4}
            LSL       r0, r3, #16
            ASR       r0, r0, #16
            BX        LR
```

With half the instruction width as standard ARM instructions, Thumb instructions are more limited in a number of ways, some of which are evident here. Compared to the ARM code, one of the most noticeable differences is that Thumb does not allow conditional execution of instructions other than branches, and every instruction sets the condition code flags (there is no S-bit in Thumb instructions). Another difference is register usage. Most Thumb instructions can only access registers `r0` to `r7`, and fewer instructions can access registers `r8` to `r12`, as opposed to ARM instructions which can access all of the general purpose registers. However, the code size will still typically be smaller due to the smaller instruction size. Also notice the `PUSH` and `POP` stack instructions, which are not part the standard ARM instruction set. Table 3 compares the code size and execution speed when using Thumb assembly versus ARM assembly for the `checksum` example.

Although we get a very noticeable difference in program size, the execution time takes a relatively large hit because there were more Thumb instructions when using `--thumb` to execute than ARM instructions when omitting `--thumb`. Moreover, the types of instructions used might not have been as efficient in Thumb.

An ARM system oftentimes employ a few different memory types and sizes. In these systems it is common for the software to be a mix of 32-bit ARM and 16-bit Thumb segments (again all specified by the compilation tools).

	Execution Time (microseconds)	Program Size* (bytes)
Compiled with `-Ospace`, `-O3`	2.467	134
Compiled with `-Ospace`, `-O3`, `--thumb`	3.050	104

*Program size reflects object file that includes **`main`** and all associated data, but not startup code.

Table 3: **`Checksum`** *simulation comparisons using* **`--thumb`**.

Interprocedural Optimizations

Interprocedural optimizations [10] differ from other compiler optimizations in that they are performed by analyzing the entire program or file as opposed to individual functions or blocks of code. One example that we have already mentioned is inline function expansion [8]. Other general examples include removing non-executing and/or redundant code (also known as "dead code elimination"), simplifying loops, and using memory more efficiently.

Many interprocedural optimizations have to do with optimizing function calls. A function call can be considered to be a "tail-call" if it is the last meaningful thing done by the calling function (other than the function's return instruction). Tail-call optimization can eliminate unnecessary returns and stack accesses in function hierarchies. Consider the following C code:

```
int foo() {
  return 1;}

int bar() {
  int x = foo();
  return x; }

int main() {
  int y = bar();
  return 0; }
```

Tail-call optimization is only done by the ARM C compiler when using `-O1` or higher. When compiling this with `-O0`, the compiler produces the following assembly:

```
foo     MOV     r0, #0x0000001
        BX      r14
bar     STR     r14, [r13, #-0x0004]!
        BL      foo
        MOV     r1, r0
        MOV     r0, r1
        LDR     r14, [r13], #0x0004
        BX      r14
main    STR     r14, [r13, #-0x0004]!
        BL      bar
        MOV     r2, r0
        MOV     r0, #0x00000000
        LDR     r14, [r13], #0x0004
        BX      r14
```

Compare this with the output using `-O1`:

```
foo     MOV     r0, #0x0000001
        BX      r14
bar     B       foo
main    STR     r14, [r13, #-0x0004]!
        BL      bar
        MOV     r0, #0x00000000
        LDR     r14, [r13], #0x0004
        BX      r14
```

The first noticeable difference between the outputs is that the compiler will sometimes do some insignificant and wasteful things at `-O0` according to the procedure call standard mentioned before. Understanding this is not important here, but more information can be found by exploring the standard [1]. The tail-call optimization attempts to replace branches that require return address overhead with direct branches that do not need to return. Examining the first output shows that the call to **`foo`** in **`bar`** requires that the return address register (**`r14`**) be first pushed on the stack so that the BL instruction can safely overwrite it. This is not needed with a direct branch, which can be used because there is no need to return to **`bar`** from **`foo`**. All of this save stack and code space and decrease execution time. Also of note are the stack instructions using **`r13`** (which is the register defined by the procedure call standard as the stack pointer) and the "**`!`**" signifier. The "**`!`**" indicates that the stack pointer will be updated with the new value used to calculate the address accessed.

The results of the simulation can be seen in Table 4.

The function **`foo`** still returns normally so that other functions can call it and return correctly. The transformations done by the tail-call optimization only apply to the calling function. Again, the other differences seen are the result of the compiler adhering to the procedure call standard.

	Execution Time (microseconds)	Program Size* (bytes)
Compiled with `-O0`	.217	56
Compiled with `-O1`	.050	32

*Program size reflects object file that includes **`main`** and all associated data, but not startup code.

Table 4: **`Bar`** *simulation comparisons.*

A Goal-Oriented Methodology for Using Compiler Options

It is important to find the best combination of options for a specific application during the development process. The following methodology is one way to do this and similar approaches can be taken. The first step is to analyze the goals of the option selection process and decide which criteria are most important to meet these goals. Some criteria include:

- **Execution Speed**
  Usually measured in MIPS or microseconds, it is the speed that the processor can execute instructions and is dependent upon a number of hardware and software factors.
- **Application Code Size**
  The application code size is the amount of memory required to hold the entire application code, including all associated data and

instructions. Code size can further be broken down into individual code segments, files, or functions.

- **Power Consumption**
  Many factors affect power consumption. Since most of these factors are hardware related, power consumption is not always a concern for software developers. However, studies have shown that memory accesses usually consume slightly more power than ALU operations. This has an even larger effect in multimedia and signal processing applications that handle large data arrays in nested loops. Power consumption will require actual hardware to be measured.

- **Build Time**
  Often referred to as the time it takes to complete the process of compiling and linking the entire software system, the build time depends on such factors as the application code size, host machine and network speed, license checkout type (floating vs. node-locked), as well as the number of compiler invocations and compiler options. We have seen that interfile optimization is one way to decrease built times. As well, there is usually a slight tradeoff between using more intense compiler optimizations and obtaining faster build times. Build times are usually measured by external utilities when the measurement method is not built into the compilation tools.

This list should not be considered complete by any means. It should also be noted that there are a variety of other criteria to consider for the overall optimization process (a superset of finding optimal compiler options), but these four are usually the most important criteria that can be influenced by the use of compiler options alone.

For build time, it was assumed that the host machine and network speeds, as well as the license checkout times, are constant.

Typically, the goals should be analyzed before developing the application, but this rarely happens. Once code has been developed, proven in terms of accuracy and correctness, and hand-optimized to be targeted to the particular platform, the process of testing compiler options can begin.

For the presented test case using this methodology, the same simulation tools were used (except that time simulation was done using an ARM7-based Atmel microcontroller [3] running at 40 MHz) and the test application was a slightly modified version of the Dhrystone Benchmark 2.1 [14] for C. Dhrystone is a popular benchmarking program used to compare the performance of code generated with different compilers. Dhrystone uses no floating-point and has a large focus on string handling. It is a good test program for the methodology since it can be heavily influenced by compiler options. The only modification made to the source code of Dhrystone 2.1 was to fix the number of runs at 60,000. Also, a breakpoint was placed on the last line of `main`. The modification and breakpoint were done so that the entire application from start to finish could be timed consistently with the profiler. Otherwise, Dhrystone waits for the user to input the number of runs (which will always take a variable amount of time) and ultimately never ends (as is the case for most embedded applications).

Each criterion was weighted based on its importance. For example, if execution speed is the most important, we could assign it an importance factor of 0.45, where each factor would be a percentage. The other three criteria factors would add up to exactly 0.55. For the test case, it was assumed that application code space was the most important criterion with a 0.75 importance factor, execution speed the second-most with a 0.25 importance factor, and power consumption and build time both had an importance factor of zero.

Optimization metric sets $\mathbf{M}_n = \{\mathbf{a}_n, \mathbf{b}_n, \mathbf{c}_n, \mathbf{d}_n\}$ were defined by applying any combination of a set **S** (to be defined shortly) of compiler options on an application, where:

a	= measurement of the most important criterion
b	= measurement of the second-most important criterion
c	= measurement of the third-most important criterion
d	= measurement of the least important criterion
n	= the nth combination of a set of options **S** (to be defined shortly)

All of the possible compiler options were studied and added to a preliminary test list if there was a possibility that the option could have had an effect on any of the criteria in question. For example, some compilers have options that suppress certain warnings or error messages. Using these options should not have any affect on the criteria, so were not added to the test list. The combinations that do not work together were known ahead of time. For example, typically only one level of general optimization can be accepted per compilation.

To create **S**, every one of the options from the test list was individually applied in combination with the lowest level of general optimization (`-O0` in this case) and any option that was used by default (`-Ospace` in this case) when the application was compiled. An option was added to **S** if any element from the resulting metric set $\mathbf{M}_n$ was less than the corresponding element of the metric set $\mathbf{M}_n$ created by compiling the application with only `-O0` (the lowest general level of optimization) and `-Ospace` (the only default option). Criteria whose importance factors were zero were completely ignored. In this case compiling the application with only `-O0` gave a code size of 95,200 bytes and an execution speed of 10.17196830 seconds (for 60,0000 runs of Dhrystone 2.1, as previously mentioned).

If an option depended on another option in order to be valid when creating **S**, both options for that case were used (see `--thumb` below in Table 5). The order of individual options tested should not

Compiler Option (elements of S)	Resulting $\mathbf{M}_n$\{code size$_n$, execution time$_n$, NA, NA\} By Applying Individual Option
`--apcs=interwork`	{93,664, 9.86574647, NA, NA}
`--apcs=interwork/ropi`	{93,664, 9.86574647, NA, NA}
`—bss_threshold=0`	{95,144, 10.17196037, NA, NA}
`-O1`	{94,644, 9.08366435, NA, NA}
`-O2`	{94,472, 8.64248447, NA, NA}
`-O3`	{97,368, 8.64251822, NA, NA}
`-Otime`	{95,400, 10.12394656, NA, NA}
`--split_sections`	{95,184, 10.17196668, NA, NA}
`--thumb —apcs=interwork`*	{92,632, 7.71468278, NA, NA}

*`--apcs=interworking` must be enabled to use `--thumb`.

Table 5: S for the ARM C compiler on Dhrystone 2.1.

matter. Table 5 shows **S** for the experiment on Dhrystone 2.1 along with the resulting metric sets $\mathbf{M}_n$ for each individual option. They are listed in the same order that they appear in the ARM C compiler documentation (alphabetically). Note that power consumption and build time are not calculated.

Application code size is presented in bytes and reflects the entire executable, including every generated object file. These files include the `main`, startup code, timer functions, library code, and all associated data. Execution times are in seconds, reflecting 60,000 runs of the Dhrystone 2.1 modules, and were recorded using the highest precision that the simulator allows. Although some of the differences in execution times might seem insignificant at first glance, they become more and more significant over time in real-life applications.

The `--apcs=interwork` option generates code with ARM/Thumb interworking and results in a significant improvement on code size when used with the `--thumb` option. The `--apcs=interwork/ropi` option enables read-only position independent code, but the interworking alone only has an impact at `-O0`. The `--bss_threshold=0` option dictates where global data eight bytes or less is placed in memory, sometimes saving the number of needed base pointer registers to access that data. The `--split_sections` option tells the compiler to generate individual ARM image-defined code sections for every function of the source code, which would only lessen the code size when used with `-O0` [2].

Once the creation of **S** was complete, every possible combination of the elements in **S** that included the default or higher level of general optimization was applied when the application was compiled. In this test case that was every possible combination of the elements of **S** that included the `-O2` or `-O3` options. The $\mathbf{M}_n$ for each of these combinations was recorded. In all practical cases at least the default level of general optimization will be needed to find the best combination of options.

Once the results were recorded, a scale factor needed to be determined for each relevant criterion **a**, **b**, **c**, and **d**. A scaling factor was not needed for a criterion whose importance factor was zero. Let us call criterion **CR** the criterion toward which other criteria were scaled (**CR** will always have a scaling factor of one). To calculate the scale factor for any other criterion **N**, the average value of **N** and **CR** (both rounded to the nearest precision of the recorded data) was obtained. Then, the average value of **N** was divided by the average value of **CR**. In this experiment, the criteria scaled toward execution time. The average execution time (rounded to the same precision as our criterion data) was 6.81108871 seconds and the average application code size was 93,030, resulting in a scale factor of .00007321 for code size. This scale factor was also rounded to the same precision as the criterion data. Scale factors for the other criteria were calculated in the same way using the same CR. In this experiment there was only one scale factor to calculate.

An overall value or "score" ($\mathbf{V}_n$) was determined for each set $\mathbf{M}_n$ using importance factors and scale factors with the following equation:

$$\mathbf{V}_n = \mathbf{a}_n\mathbf{X1Y1} + \mathbf{b}_n\mathbf{X2Y2} + \mathbf{c}_n\mathbf{X3Y3} + \mathbf{d}_n\mathbf{X4Y4}$$

where

X1 = scale factor for a
X2 = scale factor for b
X3 = scale factor for c
X4 = scale factor for d

Y1 = importance factor for a
Y2 = importance factor for b
Y3 = importance factor for c
Y4 = importance factor for d

The $\mathbf{V}_n$ for each of these combinations that was the smallest in magnitude (there could have been more than one) represented the $\mathbf{V}_n$ for the most optimal compiler option combination based on the goals according to the methodology.

Table 6 shows the best ten compiler option combinations for the experiment based on $\mathbf{V}_n$.

It should be no surprise that the most optimal combinations depend on the optimization goals. These combinations usually contain a large number of options since the default options alone try to maintain a balance between the debug view and performing optimization. For this case, using Thumb code had a large impact on code size, shown by the fact that the `--thumb` option was a part of every combination in the top ten.

Because of the heavier emphasis on speed versus code size, the best possible combination for pure speed ended up being the best overall combination for producing $\mathbf{V}_n$. Conversely, the best possible combination for pure code size ranked 41 out of the 80 combination candidates (the candidates being only those combinations that included `-O2` or `-O3`).

A high-level flow chart for the presented methodology is shown in Figure 2.

Conclusion

The only way to know the optimal combination of options is to test them intelligently based on a set of goals and how important those goals are in relation to each other. This is precisely what the presented methodology does. However, its weakness lies in the fact that it is

Compiler Option Combination	Code Size (a)	Execution Speed (b)	$\mathbf{V}_n$
`--bss_threshold=0 -O3 -Otime --split_sections --thumb --apcs=interwork`	92,224	5.29249510	6.38691306
`-O3 -Otime --split_sections --thumb --apcs=interwork`	92,280	5.29250382	6.38999006
`--apcs=interwork/ropi --bss_threshold=0 -O3 -Otime --split_sections --thumb`	92,400	5.29249525	6.39657681
`--apcs=interwork/ropi -O3 -Otime --split_sections --thumb`	92,456	5.29250337	6.39965366
`--bss_threshold=0 -O3 -Otime --thumb --apcs=interwork`	92,472	5.29249615	6.40053038
`--bss_threshold=0 -O2 -Otime --split_sections --thumb --apcs=interwork`	92,232	5.34799947	6.40122841
`-O3 -Otime --thumb --apcs=interwork`	92,528	5.29250487	6.40360738
`-O2 -Otime --split_sections --thumb --apcs=interwork`	92,288	5.34800820	6.40430541
`--bss_threshold=0 -O2 -Otime --thumb --apcs=interwork`	92,360	5.34800052	6.40825683
`--apcs=interwork/ropi --bss_threshold=0 -O3 -Otime --thumb`	92,648	5.29249600	6.41019406

Table 6: Ten most optimal option combinations for methodology applied to modified Dhrystone 2.1.

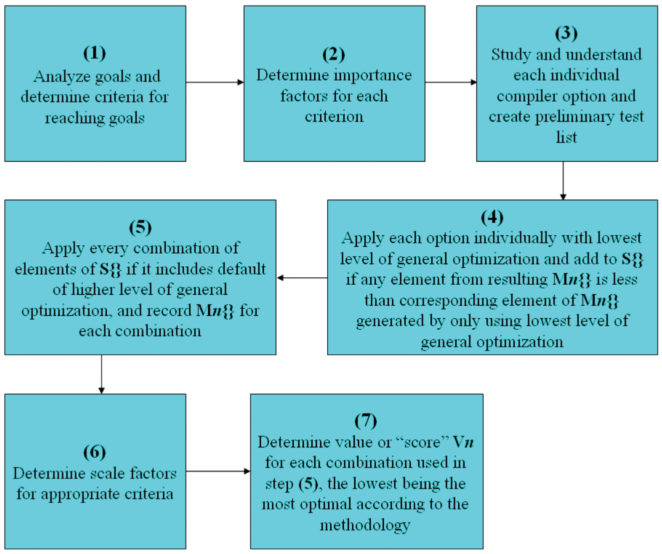
(1)
Analyze goals and determine criteria for reaching goals
(2)
Determine importance factors for each criterion
(3)
Study and understand each individual compiler option and create preliminary test list
(4)
Apply each option individually with lowest level of general optimization and add to S{} if any element from resulting Mn{} is less than corresponding element of Mn{} generated by only using lowest level of general optimization
(5)
Apply every combination of elements of S{} if it includes default of higher level of general optimization, and record Mn{} for each combination
(6)
Determine scale factors for appropriate criteria
(7)
Determine value or "score" Vn for each combination used in step (5), the lowest being the most optimal according to the methodology

Figure 2: Flow chart for finding optimal combination of compiler options.

difficult to assign importance factors when it is not known exactly how much impact compiler options alone will have on an individual criterion over another. For example, a developer might think code size is more important than execution speed before applying the methodology, but it might be the case that the percentage of possible improvement of execution speed through compiler options alone is significantly larger than the possible percentage of improvement on code size, thus influencing the importance factors of the two.

Applying compiler options is in no way an exact science. It is not always guaranteed that `-Otime` will generate faster code and `-Ospace` will generate smaller code when applied to small, simple functions. Additionally, it is not always possible to fully anticipate how a compiler translates a particular piece of source code. Further, some code optimization issues have been shown to be NP-complete, making optimizing ever-more difficult.

Still, no matter how powerful a compiler's optimization ability is, or how unpredictable it can be, it is still the job of the developer to efficiently harness its strengths and minimize its weaknesses to produce optimal code.

References

1. ARM, Ltd. 2007. *Procedure Call Standard for the ARM Architecture*. ARM Ltd., Cambridge, UK.

2. ARM, Ltd. 2007. *RealView Compilation Tools for uVision v3.1 Compiler Reference Guide*. ARM Ltd., Cambridge, UK.

3. Atmel. 2001. Atmel's ARM-based microcontroller—Low power for portable systems. White Paper. Atmel, San Jose, CA.

4. Faggin, F., Hoff, M. E., Mazor, S., and Shima, M. 1996. The history of the 4004. *IEEE Micro 16*, 12. 10-20.

5. Furber, S. 2000. *ARM System On-chip Architecture*. Addison-Wesley Professional, Harlow, UK.

6. Hall, S. 1996. *Journey to the Moon: The History of the Apollo Guidance Computer*. American Institute of Aeronautics & Astronautics, Reston, VA.

7. Hennessy, J. and Patterson, D. 2002. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Francisco, CA.

8. Lupers, R. 2000. *Code Optimization Techniques for Embedded Processors*. Kluwer Academic Publishers, Dordrecht, The Netherlands.

9. Martin, T. 2005. *Insider's Guide to Philips ARM7 Microcontrollers*. Hitex (UK) Ltd., Coventry, UK.

10. Muchnick, S. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, CA.

11. Seal, D. 2001. *ARM Architecture Reference Manual* (2nd ed.). Addison-Wesley, Harlow, UK.

12. Shen, J. 2004. *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw-Hill, New York, NY.

13. Sloss, A., Symes, S., and Wright, C. 2004. *ARM Architecture Reference Manual* 2nd Ed. ARM System Developer's Guide. Morgan Kaufmann, San Francisco, CA.

14. York, R. 2002. Benchmarking in context: Dhrystone. White Paper. ARM Ltd., Cambridge, UK.

Biography

Joe Bungo (joe.bungo@arm.com) joined ARM in 2002 as an intern in the Applications Engineering group supporting and providing training for ARM software development tools. In 2005, he joined ARM's University Relations group as an applications engineer encouraging and supporting the use of the ARM architecture in academia in all facets, including providing hands-on workshops, lectures on various ARM-related topics, and technical support. Bungo has a BA in computer science from the University of Texas at Austin.

Applied Signal Technology, Inc.

$5,000 sign-on bonus regarding these featured jobs!

We are a recognized leader in the telecommunications field and a pioneer in the development of advanced signal collection and processing equipment. We design, develop, and market digital signal processing equipment to collect and process a wide range of telecommunications signals for digital, cellular, and satellite transmissions. AST is headquartered in Sunnyvale, California, with offices in Oregon, Utah, Maryland, Virginia, Texas, Utah, Southern California, Florida, and Ohio. AST offers employees top-tiered compensation and the opportunity to develop leading edge technologies.

Apply online at:

www.appsig.com

Career Opportunities:

- **Telephony Communications**
- **Wireless Communications Systems**
- **Multichannel Systems**
- **Electronic Systems**
- **Sensor Signal Processing**
- **Oceanic Sensory Systems**
- **National Security Systems**
- **Homeland Defense**

We are proud to be an EEO/AA employer M/F/D/V.
The ability to obtain a U.S. Government security clearance is required.