Application Report
SPRAA14 − April 2004

Introduction to Compiler Consultant

George Mock

Software Development Systems

ABSTRACT

C and C++ are very powerful and expressive programming languages. Even so, these languages lack the power to express certain programming details that can be critical to achieving good compiler performance on a DSP. The role of Compiler Consultant, found in Code Composer Studio™ v3.0, is to recognize such situations, inform the developer about what is missing, and exactly how to address it. It is as if a compiler expert pores over the code and shows how to improve it specifically for a TI DSP.

This DSP compiler expertise is available to all developers. One need not be an expert to get good results from Compiler Consultant.

Performance improvements obtained through use of Compiler Consultant can be substantial. For a single loop, the code may run as much as 30 times faster. For the MP3 decoder example for the TMS320C64x™ in this application note, performance increases by over 10 times.

Contents

Trademarks are the property of their respective owners.

List of Figures

List of Tables

1 Introduction

Code Composer Studio is TI’s flagship product for developing and debugging code for execution on TI processors. Code Composer Studio v3.0 introduces a suite of tools aimed at a later part of the product life cycle: Application Code Tuning (ACT). Compiler Consultant is one of the ACT tools.

The role of Compiler Consultant is to recognize cases where the C/C++ source code fails to inform the compiler about details critical to good performance. Consultant then issues advice on how to fill the information gap. It is as if a compiler expert pores over the code and shows how to improve it specifically for a TI DSP.

Note: *Compiler Consultant and Consultant are used interchangeably.*

This DSP compiler expertise is available to all developers. One need not be an expert to get good results from Compiler Consultant, even though experts will benefit as well.

Performance improvements gained by using Compiler Consultant can be considerable. A single loop may run 30 times faster. Applying the advice generated for the C64x™ MP3 decoder example in this application note improves performance by over 10 times.

This performance improvement is attained, on a small scale, in a few minutes. Improvement across an entire application is often seen in a few hours. That is because Compiler Consultant focuses on the exact points which need attention, and issues specific advice on what to do. Achieving similar results without the aid of a tool like Compiler Consultant would likely take much longer.

TI compilers contain many features dedicated to improving performance. These features are well documented in the manuals and online help. Even so, it can be difficult to understand where or why to apply these features. Compiler Consultant, on the other hand, recommends use of a performance feature in context of a specific optimization opportunity in the code. As such, the where of using the feature is made obvious by Consultant, and the why is explained in the Consultant advice.

At this writing, Compiler Consultant is available only for the C6000™ compiler. Support for other TI compilers will be released later.

2 Not Just for Experts

One major thrust of Consultant is to make DSP compiler expertise available as widely as possible. By design, then, one need not be a compiler or DSP expert to get good results from Consultant. Experts stand to gain as well, in that Consultant can be applied to the parts of the code which have not yet been tuned.

Consultant advice often requires the user to understand the code well enough to guarantee some condition is always met. For instance, Consultant may advise use of the MUST_ITERATE pragma to indicate a loop always iterates some minimum number of times. Consultant will show exactly where and how to use the pragma. The developer’s role is to determine whether the condition regarding the minimum number of loop iterations is always met.

Although some advice may be obvious to expert developers, other advice is difficult even for experts to determine on their own.

The performance improvements which come from following Consultant advice are sometimes complicated. It is not necessary to understand these improvements in full detail. This application note explains such improvements to motivate the gathering and implementation of the advice, and not to compel any developer to become an expert in the details.

Sometimes following a single piece of advice will result in no improvement. In such a case, it may be necessary to implement two or three pieces of advice in succession before improvement is realized.

3 Consultant Targets Loops

The C6000 is a very long instruction word (VLIW) architecture that can execute up to 8 instructions in parallel. Utilizing this parallelism is central to achieving peak performance. First among the techniques for exploiting instruction parallelism in C/C++ code is software pipelining the loops. In software pipelining, multiple iterations of a loop are executed simultaneously in software. Consider the following loop, written in assembly level pseudo-code.

```
loop:
       load
       operate
       store
       bdec loop, count             ; decrement count, branch if count != 0
```

Because there is no parallelism, only one instruction is executed each cycle. The number of cycles for this loop is 4 * count.

Here is a simplified view of the software pipeline for this loop. The (n) represents the iteration of the original loop for that instruction. Note instructions combined with the “||” operator are executed in the same cycle, or in parallel.

```
      load(n)                                                  ; loop prolog
      load(n+1) || operate(n)
      load(n+2) || operate(n+1) || store(n)
loop:
      load(n+3) || operate(n+2) || store(n+1) || bdec loop, count(n)  ; loop kernel
                   operate(n+3) || store(n+2)                  ; loop epilog
                                   store(n+3)
```

Much more work is done per cycle by doing things in parallel. This loop runs in count + 2 cycles, approximately four times faster than the original loop.

More information on software pipelining can be found in the *TMS320C6000 Programmer's Guide* (SPRU198). Keep in mind, however, that detailed knowledge of software pipelining is not required to get good results from Consultant.

Compiler Consultant targets loops because loops present the best opportunity to improve performance. Effective software pipelining requires a great deal of loop analysis information. In many cases, not all of the information required is available in the source code. Compiler Consultant analyzes the loops in the application in order to determine what information may be missing, and then issues advice on how to address the information gap. Code outside of loops is not analyzed by Compiler Consultant.

4 Types of Advice

4.1 Options Advice

Sometimes the consultant advice is to simply turn on, or off, a compiler build option. A compiler build option controls how the compiler works. For example, −g enables the generation of additional information needed to debug the code.

Section 6.6 of the MP3 example describes a common scenario with options advice. A project which uses the default Code Composer Studio options builds with the optimization option –o*n* disabled, and the debug option –g enabled. When debugging, those are generally the correct options to use. When tuning for performance, optimization is much more important. That is why the advice reminds a developer to reverse those options: enable optimization and disable debug. Using optimization enables many levels of compiler analysis that lead to vastly improved performance. Disabling debug further improves performance, because the presence of debug information inhibits optimization. In general, other advice is not generated without first following this options advice.

Some options include a numeric argument. Examples include the speculative load threshold option –mh*n* and the automatic inlining threshold option –oi*n*. Advice for using these options includes both the details for using them properly, and the needed value of n.

The performance impact of options advice depends as much on the original options as the new options in the advice. If, as in the MP3 example, the change is from no optimization to full optimization, the impact is very large, a more than 6X increase in speed. Simply adding a –mh*n* or –oi*n*, however, will tend to have less impact.

4.2 Alias Advice

When two or more variables can refer to the same memory address, they are called aliases. The presence of presumed aliases sharply degrades the performance of a loop. Following alias advice informs the compiler aliases are not present, which typically results in a significant performance increase.

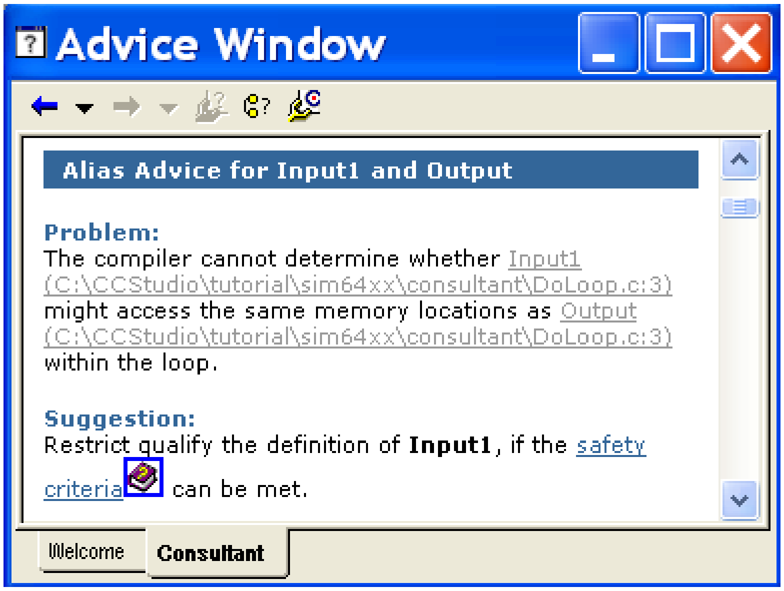
Consider the software pipelining example in section 3. Note how the load and store instructions change order. In the original loop, store(n) always occurs before load(n+1). Not so in the software pipelined loop. Note load(n+1) occurs before store(n), and load(n+2) occurs at the same time as store(n). If the load and store references are not aliases, i.e. they never refer to the same memory address, then such reordering works. If the load and store references are aliases, reordering does not work. If the compiler cannot prove the load and store references are not aliases, it must presume they could be aliases. For this example loop, that forces every store(n) to occur before load(n+1), which almost completely disables software pipelining.

Alias advice includes exactly which variable(s) are possible aliases, and how to use the restrict keyword to indicate those variable(s) are not aliases.

Consider the loop in this function.

```
void DoLoop(short *Input1, short *Input2, short *Output,
            short *Weights, int Count)
{
    int i, Vector1, Vector2;
    short Weight1 = Weights[0];
    short Weight2 = Weights[1];
    for (i = 0; i < Count; i++)
    {
        Vector1 =  Input1[i] * Weight1;
        Vector2 =  Input2[i] * Weight2;
        Output[i] = (Vector1 + Vector2) >> 15;
    }
}
```

One of two pieces of alias advice is shown in Figure 1.



Alias Advice for Input1 and Output

Problem:
The compiler cannot determine whether Input1 (C:\CCStudio\tutorial\sim64xx\consultant\DoLoop.c:3) might access the same memory locations as Output (C:\CCStudio\tutorial\sim64xx\consultant\DoLoop.c:3) within the loop.

Suggestion:
Restrict qualify the definition of **Input1**, if the safety criteria can be met.

Figure 1. Alias Advice

The suggestion is to use the restrict keyword on the variable Input1, provided the safety criteria can be met. Note the words **safety criteria** are a link to online help which defines the safety criteria for using restrict. In this case, Input1 must be guaranteed to be the only pointer accessing its range of memory locations. The other piece of advice, not shown, indicates that Input2 and Output are aliases. Since Output is part of both alias pairs, applying the restrict keyword to Output resolves both pieces of advice.

The performance impact of removing aliases can be enormous. Measured on a single loop, the improvement can be as much as 30X, or higher.

A comprehensive example of alias advice is contained in section 6.9.

4.3 Trip Count Advice

The term trip count refers to how many times a loop iterates. Following trip count advice allows the compiler to make a loop smaller, and improve how the loop is optimized with the surrounding code.

Consider, once again, the pseudo-code example of a software pipelined loop in section 3. Note that by the time the first decrement and branch executes, 4 iterations of the loop have started. This code works only if the trip count of the original loop is at least 4.

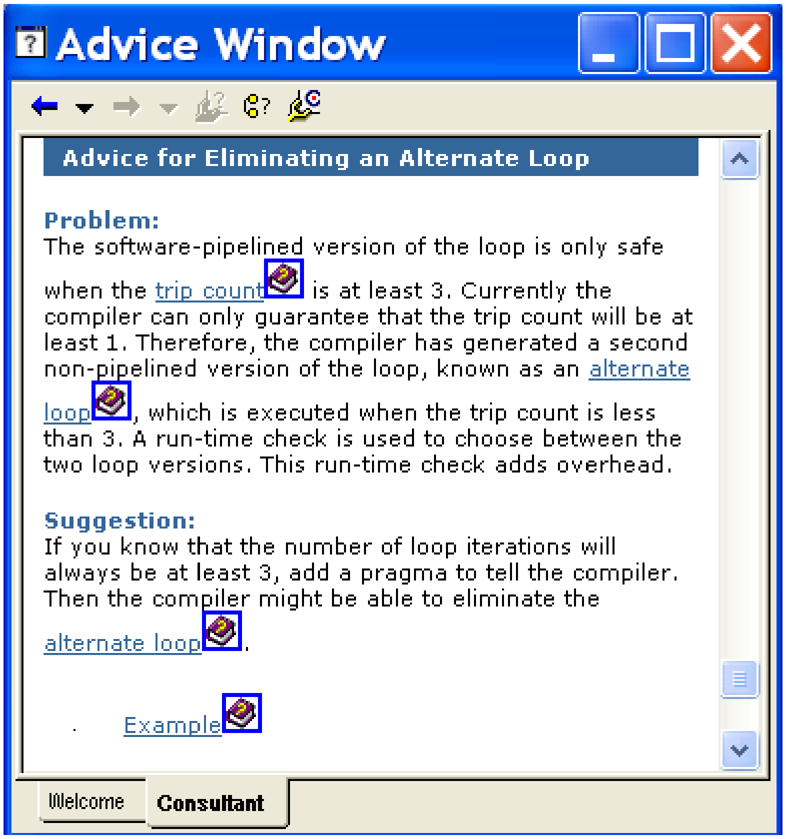
In absence of other information, the compiler generates code similar to the following …

```
if (count >= 4)
{
   loop prolog
   loop kernel
   loop epilog
}
else
   alternate loop
```

The alternate loop is not software pipelined, and thus has no restrictions for a minimum trip count.

Trip count advice is aimed at eliminating both the alternate loop and the branches to perform the runtime check on the trip count. Trip count advice calls for using the MUST_ITERATE pragma or the –mh*n* option.

For the example loop in section 4.2, Figure 2 shows a piece of trip count advice.



Advice for Eliminating an Alternate Loop

Problem:
The software-pipelined version of the loop is only safe when the trip count is at least 3. Currently the compiler can only guarantee that the trip count will be at least 1. Therefore, the compiler has generated a second non-pipelined version of the loop, known as an alternate loop, which is executed when the trip count is less than 3. A run-time check is used to choose between the two loop versions. This run-time check adds overhead.

Suggestion:
If you know that the number of loop iterations will always be at least 3, add a pragma to tell the compiler. Then the compiler might be able to eliminate the alternate loop.

Example

Figure 2. Trip Count Advice

The advice indicates that if the compiler knows the trip count is at least 3, then it will not generate the alternate loop. The **Example** link shows how to use the MUST_ITERATE pragma.

In some cases, such as those in section 6.10, the actual minimum trip count is below the threshold given in the advice. In those cases, it is often possible to use the speculative load option –mh*n*. This option tells the compiler it is acceptable to read (not write) n bytes past the beginning or end of an array or memory block. The value of n required is given in the advice. When the compiler can issue such speculative loads, it can often structure the software pipeline to work with any trip count, and thus eliminate the alternate loop.

To understand how following trip count advice can transform a loop, compare the following pseudo-code example with the one at the beginning of this section. Note the removal of both branches and the alternate loop. This removal is signaled by putting a “;” at the beginning of a line to comment it out. Note how those changes open up other opportunities for optimization.

```
;       if (count >= 4)         ; conditional branch removed
;       {
           loop prolog          ; optimized with code before the loop
           loop kernel
           loop epilog          ; optimized with code after the loop
;       }
;       else                    ; unconditional branch removed
;          alternate loop       ; entire loop body removed
```

Following trip count advice saves both code space and cycles. Removing the alternate loop saves code space. The larger the original loop, the more space is saved. Cycles saved can speed up a single loop by as much as 7X. Cycles are saved in two ways. The two branches for controlling which loop is executed, one conditional and one unconditional, are removed. That change can save as much as 10 cycles. Further cycles are saved because the loop prolog and epilog are no longer bracketed by the branching code, and thus can be optimized with the code adjacent to the loop. On a loop the application enters many times, these cycle savings can be significant.

4.4 Alignment Advice

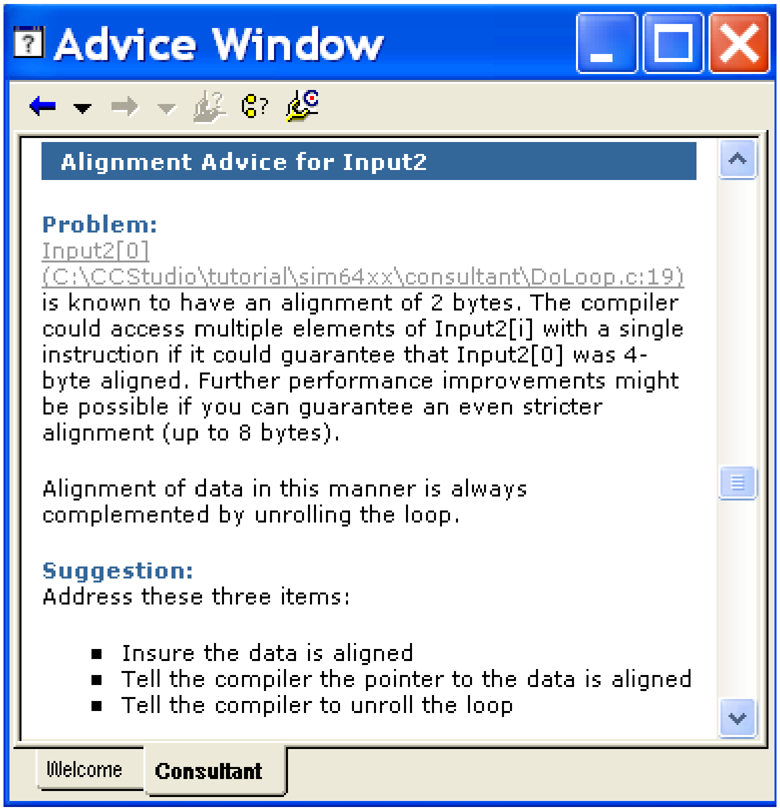
Following alignment advice allows the compiler to use special instructions which operate on multiple array elements. This is a deeper level at which code can perform multiple operations at the same time. It is called alignment advice because one of the key steps is to align the data on a certain address boundary. The special instructions utilize single instruction multiple data (SIMD, pronounced “sim-dee”).

For instance, one add instruction uses two 16-bit shorts as operands. Those short operands, in addition to being adjacent, must be aligned on a 4-byte boundary. Because this alignment is greater than that normally provided for shorts, additional steps are required.

Figure 3 shows a piece of alignment advice for the loop from section 4.2. Consultant advice states the variable Input2 would benefit from being aligned. An online help entry describes how to align the data. Another help entry shows how to use _nassert to inform the compiler about the alignment. Finally, the advice shows how to use the MUST_ITERATE pragma to indicate the trip count is always some multiple of m, and thus the loop can be unrolled m times.

Following alignment advice can speed up a loop by as much as 8X. However, improvements of 2X and 4X are much more common. The C64x™ device has many more SIMD instructions than the C62x™ or C67x™. In particular, some C64x SIMD instructions can access single byte elements. Thus, alignment advice is much more common on C64x than C62x or C67x devices.

Section 6.11 shows how alignment advice is used in the MP3 example.



Alignment Advice for Input2

Problem:
Input2[0] (C:\CCStudio\tutorial\sim64xx\consultant\DoLoop.c:19) is known to have an alignment of 2 bytes. The compiler could access multiple elements of Input2[i] with a single instruction if it could guarantee that Input2[0] was 4-byte aligned. Further performance improvements might be possible if you can guarantee an even stricter alignment (up to 8 bytes).

Alignment of data in this manner is always complemented by unrolling the loop.

Suggestion:
Address these three items:

- Insure the data is aligned
- Tell the compiler the pointer to the data is aligned
- Tell the compiler to unroll the loop

Figure 3. Alignment Advice

4.5 Order of Advice Implementation

Section 6.7 details how to find the most important pieces of Consultant advice by sorting on profile data or a compiler generated estimate of performance. When considered by advice type, however, the typical application will achieve the best performance for the time invested by implementing the advice in this order:

- Options advice
- Alias advice
- Trip count advice
- Alignment advice
- All remaining advice

5 Usage Overview

This section is a quick overview of how to use Compiler Consultant. In addition to being a good reference, it serves as guide to how Consultant is applied to the MP3 example.

1. Build with the −−consultant option.

   This build option instructs the compiler to analyze your source code and build options. The resulting advice is stored in specially coded information files. The rest of Compiler Consultant presents this advice so you can easily access and apply it.

2. Optionally, profile the CPU performance of your loops.

   This runtime loop information can be used to prioritize which pieces of advice to implement first.

3. Launch the Profile Viewer to inspect the consultant advice, and, if available, the CPU runtime loop information.

   Unlike other tuning tools, Compiler Consultant does not have a dedicated graphical user interface. Rather, the summary advice is displayed in another new tuning tool, the Profile Viewer. Profile Viewer displays profile results, and Compiler Consultant advice, in a tabular format very similar to a spreadsheet. Each row represents one loop in the application. Note Consultant does not issue advice for code outside of loops.

4. Typically, there are many loops. To determine which are most important, sort by double clicking on the relevant column heading.

   If you did the profiling of step 2, you can sort on that data to see which loops took the most cycles. If you did not profile, you can sort the loops by a compiler generated estimate of how many cycles it takes for a single iteration.

5. To see advice for a specific loop, double-click on the Advice Count or Advice List cell in that row.

6. The advice displays in the Advice Window. Click on links to navigate through parts of the advice, or to related help entries, or to open the source file on the lines which contain the loop.

   The Advice Window is another of the new tuning tools.

7. Implement as much or as little of the advice as you choose.

8. Build again to see the improvements.

Compiler Consultant does not limit you to building within Code Composer Studio. You can also build outside of Code Composer Studio and import the Compiler Consultant information for analysis. Any such builds must use the −−consultant build option. The consultant information files can then be imported. For more information access: **Help | Contents | Application Code Tuning | Compiler Consultant | Consultant Usage Without Building in Code Composer Studio | Import**.

6 Example Based on MP3 Decoder

6.1 Overview

The purpose of this extended example is twofold. The first purpose is to illustrate the performance improvement Compiler Consultant can achieve on an entire application. In this case, the speed up is over 10 times! The second purpose is to show how straightforward it is to gather the advice and implement it.

The example project is an MP3 decoder. The source code is presented in two projects. The first one is called mp3. At this point, the code is working but has not been tuned. No Compiler Consultant advice has been generated or applied. The second project is mp3_tuned. This is the state of the project after much of the Compiler Consultant advice has been applied. The example shows how Compiler Consultant guides you through the process of modifying the initial project to become the tuned project.

This example was developed using the C6416™ device simulator as the execution platform. If you use a different execution platform, you will see some differences with what is described here. In particular, if you choose a C62x or C67x platform, you will get different advice.

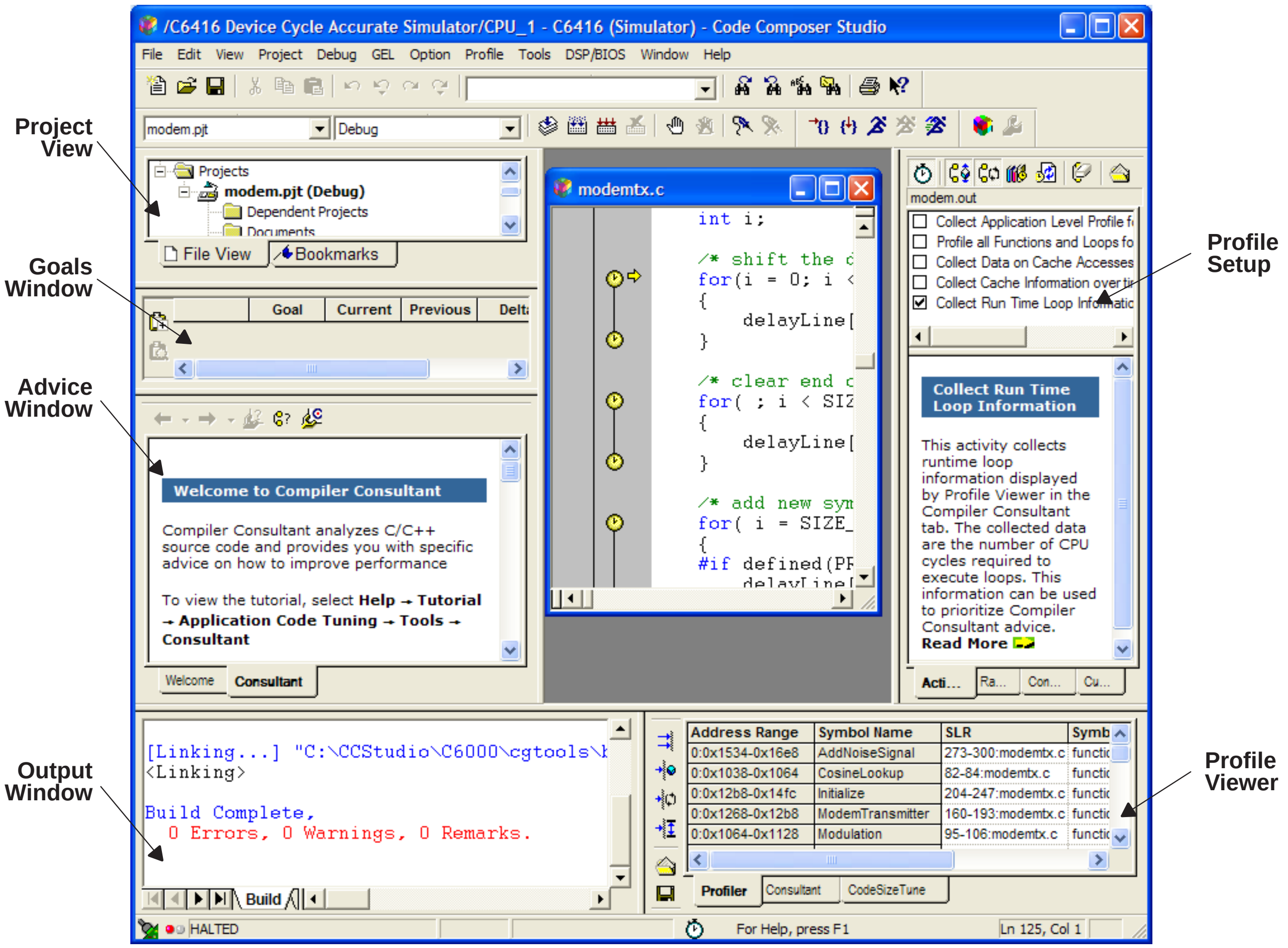
6.2 General Directions

Except for Getting Started, each set of directions presumes Code Composer Studio is in Tuning Layout, the mp3 project is open and active, and the application executable mp3.out is loaded. Any execution of the decoder to collect profile data presumes profiling is enabled, and the Halt/Resume profile collection points described in section 6.4 are in place.

When the directions state “Launch *tool name,*” that can be done one of two ways. The Advice Window may contain a clickable icon, or each tool can be launched by selecting it from the **Profile** menu, or the **Tuning** sub-menu under **Profile**.

Paragraphs that begin with **TIP:** are brief departures from the main thread to impart some insider knowledge. Some tips are specific to Compiler Consultant, but many are not.

TIP: When ACT tools launch, they dock along the sides of the display. When a window is in the docked state, it is not framed with a title bar identifying the window. Thus, a new user may lose track of which window is which. As a general rule of thumb, here is where these windows dock, starting on the upper left and moving counter-clockwise: Project View, Goals Window, Advice Window, Output Window, Profile Viewer, Profile Setup (see Figure 4). To see the name of a window, right-click and choose Float in Main Window. Among other changes, the window is framed with a title bar which identifies it. All the figures show the windows in this state. To re-dock a floating window, right-click and choose Float in Main Window.



Welcome to Compiler Consultant

Compiler Consultant analyzes C/C++ source code and provides you with specific advice on how to improve performance

To view the tutorial, select **Help → Tutorial → Application Code Tuning → Tools → Consultant**

```
int i;

/* shift the c
for(i = 0; i <
{
    delayLine[
}

/* clear end c
for( ; i < SIZ
{
    delayLine[
}

/* add new sym
for( i = SIZE_
{
#if defined(PF
    delayLine[
```

☐ Collect Application Level Profile f
☐ Profile all Functions and Loops fo
☐ Collect Data on Cache Accesses
☐ Collect Cache Information over ti
☑ Collect Run Time Loop Informatic

Collect Run Time Loop Information

This activity collects runtime loop information displayed by Profile Viewer in the Compiler Consultant tab. The collected data are the number of CPU cycles required to execute loops. This information can be used to prioritize Compiler Consultant advice.
Read More

```
[Linking...] "C:\CCStudio\C6000\cgtools\b
<Linking>

Build Complete,
  0 Errors, 0 Warnings, 0 Remarks.
```

Address Range	Symbol Name	SLR	Symb
0:0x1534-0x16e8	AddNoiseSignal	273-300:modemtx.c	functic
0:0x1038-0x1064	CosineLookup	82-84:modemtx.c	functic
0:0x12b8-0x14fc	Initialize	204-247:modemtx.c	functic
0:0x1268-0x12b8	ModemTransmitter	160-193:modemtx.c	functic
0:0x1064-0x1128	Modulation	95-106:modemtx.c	functic

Figure 4. Typical Tuning Layout

6.3 Getting Started

Use these steps to insure the mp3 project opens, builds, loads, and runs properly.

1. Unzip the contents of mp3_projects.zip into an empty directory.

   This step creates two CCS project directories: mp3 and mp3_tuned.

2. Bring up CCStudio.
3. Under the **Project** menu, choose **Open.**
4. Browse to the mp3 directory created in step 1 and choose mp3.pjt.
5. Build the project. Under the **Project** menu, choose **Rebuild All** or click this icon .
6. Load the application. Under the **File** menu choose **Load Program**, browse to the Debug sub-directory of mp3 and choose mp3.out.

7. Execute the application. Under the **Debug** menu choose **Run**, or press **F5**.

8. Several lines of output appear in the Stdout tab of the Output window. The last few lines are ...

```
Completed frame 9
Completed frame 10

Done
```

6.4 Initial Data Collection

Now that the application is written and debugged, it is time to begin tuning. Start by determining current performance. All performance comparisons are against the baseline established in this section.

1. Use the Code Composer Studio layout dedicated to code tuning. In Figure 5, the icon for Tuning Layout is circled in red. Click on it.

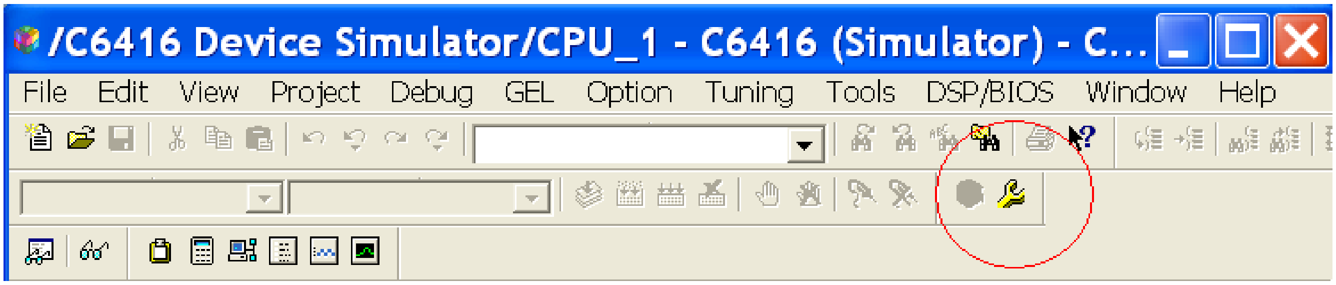

/C6416 Device Simulator/CPU_1 - C6416 (Simulator) - C...
File Edit View Project Debug GEL Option Tuning Tools DSP/BIOS Window Help

Figure 5. Tuning Layout Icon

The Advice Window is on the lower left side. It walks you through the process of tuning your code. The next several steps generally follow what is written in the Advice window, with some extra detail.

2. Click the Setup Advice icon.

   The Setup tab of the Advice Window opens.

3. Launch Profile Setup.

   Profile Setup is another of the new Tuning tools. It is the interface for selecting the type and range of profile data to collect.

4. Enable profiling by clicking on the icon in the upper left corner of the Profile Setup window.

   TIP: There are two ways to know if profiling is enabled. The enable profiling icon looks like a button that is pressed down. Also, a copy of the enable profiling icon appears at the bottom center of the Code Composer Studio display.

5. Select **Collect Application Level Profile for Total Cycles and Code Size**

6. In Profile Setup, select the **Control** tab at the bottom.

   The directions here depart from those in the Advice Window to set up Halt and Resume points for controlling exactly where profile data is collected. For the MP3 application it is best to count cycles spent in the decoder proper, not initialization and elsewhere.

7. In the Project View, expand the tree to expose the source file **towave.c**. Double-click to open that file.

8. Browse to line 188, the opening brace of the function main.

   TIP: The current line and column number of the editor cursor position is displayed in the lower right corner.

9. Highlight that entire line, then click and drag it on top of **Halt Collection** in the lower pane of Profile Setup.

   The beginning of profile collection is delayed.

10. Browse down to line 358, the primary call to the decoder.

11. Highlight that entire line, then click and drag it on top of **Resume Collection**.

    Profile collection begins at this point.

12. Highlight the next non-blank line (line 360), then click and drag it on top of **Halt Collection**.

    Profile collection ends at this point. Figure 5 shows how Profile Setup appears upon the completion of these steps . These directions resume following those in the Advice Window.

13. To insure accurate profile collection, reset the device. Under **Debug** choose **Reset**.

14. Reload the application. Under **File** choose **Reload Program**.

15. Run the program. Under **Debug** choose **Run**, or press **F5**.

16. Wait for execution to complete.

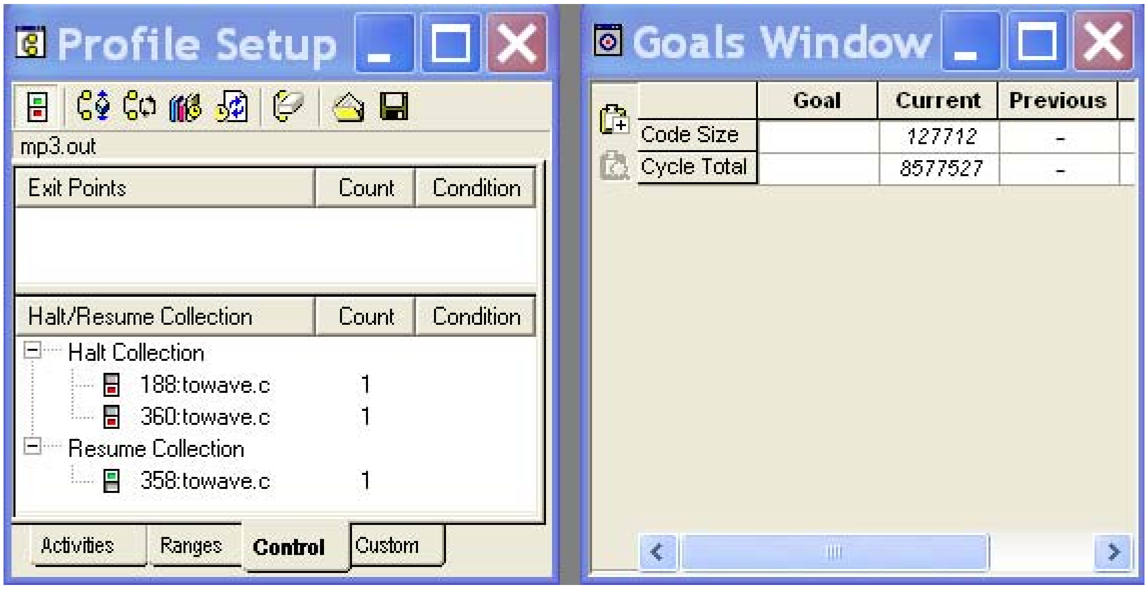
17. Do **not** launch the Profile Data Viewer at this time.

18. Click on the General Tuning Advice icon. It is near the bottom of the Setup tab in the Advice Window.

    This opens the General tab of the Advice Window.

19. Launch the Goals Window.

Figure 6 shows the Goals Window. Note it takes approximately 8.6 million cycles to run the decoder over 10 frames. It is time to see what advice Compiler Consultant has for improving this performance.



mp3.out

Exit Points	Count	Condition

Halt/Resume Collection	Count	Condition
Halt Collection		
188:towave.c	1	
360:towave.c	1	
Resume Collection		
358:towave.c	1	

	Goal	Current	Previous
Code Size		127712	-
Cycle Total		8577527	-

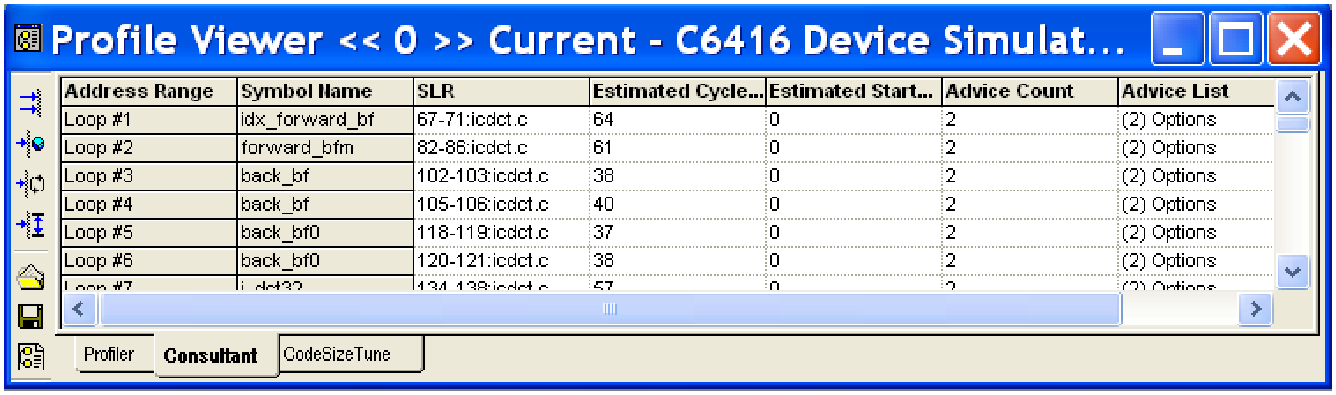
Figure 6. Profile Setup of Halt and Resume Points (left). Goals Window After First Run (right).

6.5 Build to Get Compiler Consultant Advice

1. Under the **Project** menu, choose **Build Options**…
2. In the **Build Options** dialog, click the **Compiler** tab.
3. Click on the **Feedback** item in the **Category** list.
4. Click on the checkbox **Generate Compiler Consultant Advice (−−consultant)**.

   This is the build option for generating Consultant advice.
5. Click **OK** to close the **Build Options** dialog.
6. From the **Project** menu, choose **Rebuild All** or click this icon .
7. Wait for the build to complete.
8. Launch the Profile Viewer.
9. In the Profile Viewer choose the **Consultant** tab.

   The summary of the generated advice is shown in Figure 7. Notice how every loop has two pieces of options advice.



Address Range	Symbol Name	SLR	Estimated Cycle...	Estimated Start...	Advice Count	Advice List
Loop #1	idx_forward_bf	67-71:icdct.c	64	0	2	(2) Options
Loop #2	forward_bfm	82-86:icdct.c	61	0	2	(2) Options
Loop #3	back_bf	102-103:icdct.c	38	0	2	(2) Options
Loop #4	back_bf	105-106:icdct.c	40	0	2	(2) Options
Loop #5	back_bf0	118-119:icdct.c	37	0	2	(2) Options
Loop #6	back_bf0	120-121:icdct.c	38	0	2	(2) Options
Loop #7	i_dct32	134-138:icdct.c	57	0	2	(2) Options

Figure 7. Initial Consultant Summary Advice

6.6 Implement Options Advice

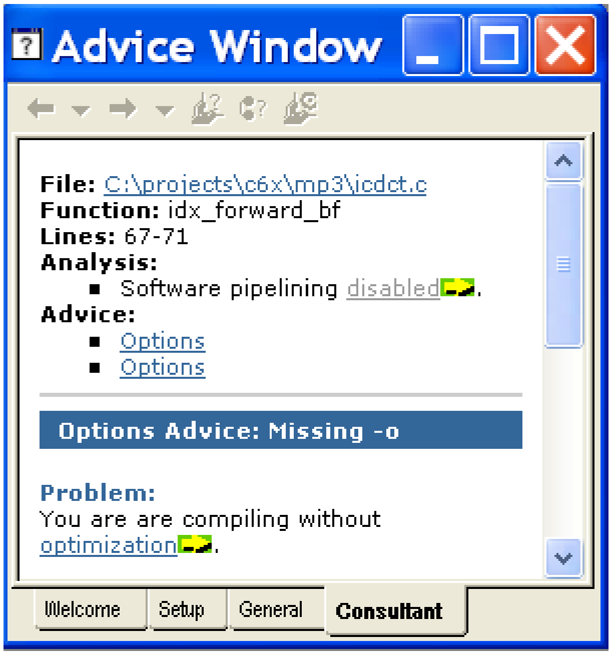
Following this advice alone makes the code run over six times faster.

1. Double-click on the Advice List cell for the first loop.

   The Consultant tab opens in the Advice Window. The advice for that loop is displayed. See Figure 8.
2. Explore the advice.

   The interface to the advice is very similar to a web browser. Links are underlined, and in blue. The **File** link brings up that source file on the corresponding loop. The **Advice** links take you to different parts of the advice. Other navigation aids include the scroll bar on the right, and the arrow icons at the top. Any compiler term is linked to an entry in online help which defines the term.

   The advice in this case states you are compiling without optimization and with –g, the debug switch. It advises enabling optimization, and not using −g. If you explored all the other loops, you would see they have the same advice. This pattern of advice occurs whenever you build without optimization or with debug. Seeing further advice requires the use of optimization and disabling debug.



File: C:\projects\c6x\mp3\icdct.c
Function: idx_forward_bf
Lines: 67-71
Analysis:

- Software pipelining disabled.

Advice:

- Options
- Options

Options Advice: Missing -o

Problem:
You are are compiling without optimization.

Figure 8. Advice for the First Loop

At this point it seems advisable to change the build options. It turns out, however, that for the file **towave.c**, this is not a good idea. Enabling optimization and disabling debug for that file makes it difficult to set up the Halt/Resume profile collection points. Since the code in **towave.c** is not part of the profile area, it doesn't affect the results to build it without optimization. Because building **towave.c** without optimization and with debug will cause Compiler Consultant to continue to issue options advice that will never be implemented, the generate consultant advice for **towave.c** will also be disabled.

The following directions show how to change the build options to those given by the advice for all the source files, then change the build options only for **towave.c**.

3. Under the **Project** menu, choose **Build Options** …
4. In the **Build Options** dialog, click the **Compiler** tab.
5. Click on the **Basic** item in the **Category** list.
6. From the **Generate Debug Info** drop down list, choose **No Debug**.
7. From the **Opt Level** drop down list, choose **File (−o3)**.
8. Click **OK** to close the **Build Options** dialog.

   The build options for the entire project are now those recommended in the advice. These next steps change the build options for **towave.c** back to the previous state, plus disable generating Consultant advice.

9. In the Project View, expand the tree to expose the source file **towave.c**. Right-click on the file and choose **File Specific Options** …
10. In the **Build Options** dialog, click the **Compiler** tab.
11. Click on the **Basic** item in the **Category** list.
12. From the **Generate Debug Info** drop down list, choose **Full Symbolic Debug (−g)**.

13. From the **Opt Level** drop down list, choose **None**.
14. Click on the **Feedback** item in the **Category** list.
15. Click to clear the checkbox **Generate Compiler Consultant Advice (−−consultant)**.
16. Click **OK** to close the **Build Options** dialog.
17. From the **Project** menu, choose **Build** or click this icon .
18. Wait for the build to complete.

Following this options advice achieves two main results. Performance increases by six times, as shown in the next section. And, much more advice is generated.

6.7 Sort the Loops

Now Compiler Consultant has generated several different types of advice for over 100 loops. This section demonstrates two different ways to determine which loops to work on first.

1. If it isn't open, launch the Profile Viewer. Click on the **Consultant** tab.

   Data in the Profile Viewer can be sorted by double-clicking on the column heading of interest.

2. Hover the mouse over the column that begins **Estimated Cycles** … to see a pop-up which shows the full name of the column is **Estimated Cycles Per Iteration**.
3. Double-click multiple times to sort the rows by that column. Each double-click toggles the sort between descending and ascending order.

   Estimated Cycles Per Iteration is a compiler generated statistic. It is an estimate of the number of CPU cycles required to complete a single iteration of the loop. It is usually the best choice for sorting the loops which doesn't require collecting runtime profile information. Note that CPU cycles do not include cycles lost to system effects such as cache misses, memory bank conflicts, etc.

   Because it doesn't account for the number of loop iterations, Estimated Cycles Per Iteration is not a certain measure of the most cycle intensive loops. In these next steps we collect runtime loop information and then compare that data with Estimated Cycles Per Iteration.

4. If it isn't already open, launch Profile Setup. Click on the **Activities** tab.
5. Make sure profiling is enabled. See step 4 of section 6.4.
6. Make sure the application executable mp3.out is loaded
7. If it is not checked, check the box for **Collect Application Level Profile for Total Cycles and Code Size**.

   Continue to use this activity to track, in the Goals Window, how much the overall cycle count goes down.

8. Check the box for **Collect Run Time Loop Information**.

   The details on this choice of activity are discussed in the next section.

9. If the Halt/Resume profile collection points described in section 6.4 are not set, set them again. Use steps 6−12.
10. To insure accurate profile collection, reset the device. Under **Debug** choose **Reset**.

11. Reload the application. Under **File** choose **Reload Program**.
12. Run the program. Under **Debug** choose **Run**, or press **F5**.
13. Wait for execution to complete.

    Notice how data in the Profile Viewer now contains a second column titled **cycle.CPU:Excl.Total**. This statistic is the number of CPU cycles, excluding system effects and calls, spent in each loop.

    The Goals Window shows a cycle count of approximately 1,330,000 cycles. The code is running over 6 times faster! This performance lift is due to the options advice implemented in the previous section. Optimization is enabled and debug is disabled.
14. Launch Profile Viewer again. Click on the **Consultant** tab.

    A second Profile Viewer window appears.
15. In one Profile Viewer window double-click on the **Estimated Cycles Per Iteration** column to sort the data by that metric. In the other Profile Viewer window, sort on the column **cycle.CPU:Excl.Total**.

    One loop from the function idx_forward_bf, and another from i_window are near the top of both windows. Otherwise, the two sorts are quite different. Whether to collect runtime profile data is a trade-off between development time and data accuracy that must be made on a case by case basis. Since running this MP3 example on 10 frames does not take much time, the directions in this application note call for collecting runtime information after implementing each round of advice.

6.8 Details on Collecting and Viewing Run Time Loop Information

If the Activities tab of Profile Setup does not display **Collect Run Time Loop Information**, then the execution platform you are using does not support it. Instead, use the activity **Profile All Functions and Loops for Total Cycles.** Details about using these two activities with Compiler Consultant are available in online help. Choose **Help | Contents | Application Code Tuning | Compiler Consultant | Setup & Build | Comparing Profile Setup Activities**.

6.9 Implement Alias Advice

Following the alias advice in this section reduces the cycle count another 28%. Overall, the code will run nine times faster.

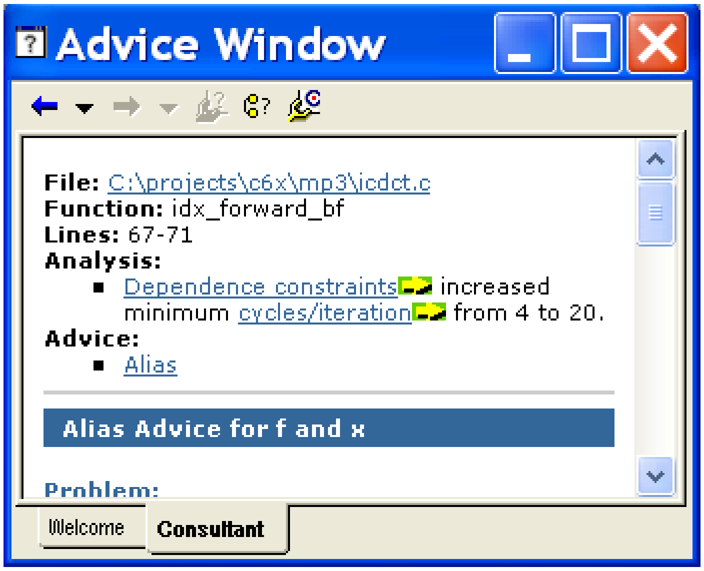
This set of directions presumes you have completed the profiling of the previous section.

TIP: To gain space on the screen, close windows that are not in use. Close floating windows by clicking on the X in the upper right corner. Close docked windows by right-clicking within the window and choosing either Close or Hide. Closing the Profile Setup window is often a good choice. It is easy to launch again as needed.

1. If you still have two Profile Viewer windows open, remove the one that has the loops sorted by **Estimated Cycles Per Iteration**. Right-click and choose **Close**.

   Now there is just one Profile Viewer window open with the data sorted by **cycle.CPU:Excl.Total**.
2. Double-click on the Advice Count or Advice List cell of the first row.

   This step opens the Consultant tab of the Advice Window, and displays the Compiler Consultant advice for this loop. See Figure 9.



File: C:\projects\c6x\mp3\icdct.c
Function: idx_forward_bf
Lines: 67-71
Analysis:
- Dependence constraints increased minimum cycles/iteration from 4 to 20.

Advice:
- Alias

Alias Advice for f and x

Problem:

Figure 9. Advice Window Inner Loop in idx_forward_bf

3. Click the **File** link to set the cursor in the editor on the loop.
4. Click the **Alias** link to see the advice.

   The advice states the compiler cannot determine whether the array **f** and the array **x** might access the same memory locations. The primary suggestion is to restrict qualify **f**, if it is safe to do so. To determine whether it is safe, first see whether the safety criteria can be met.
5. Click on the **safety criteria** link in the primary suggestion.

   Online help opens on the description of the safety criteria for using restrict. As applied to this situation, it means that the arrays passed in to **f** must be distinct from the arrays passed in to **x**. Inspection of all the calls to idx_forward_bf show that a locally declared array is always passed in as the argument to **f**, and that array is never passed to **x** in the same call**.** So, **f** does meet the safety criteria.
6. Click on the **Example** link in the primary suggestion.

   This example shows the proper way to use the restrict keyword. In this case, the keyword must be placed inside the brackets of the declared array …

   ```
   static void idx_forward_bf(int m, int n, INT32 x[], INT32 f[restrict],
                              DCTCOEF coef[])
   ```
7. Change the source to use the restrict keyword. Use control-S to save the file. If there were a prototype for this function, it would need to be modified as well.
8. The second row is a loop in the function unpack_samp which has no advice.
9. The third and fourth rows are about two loops in the function i_window. Explore the advice for both loops.

   There is alias advice for one loop, and the recommendation is to use the restrict qualifier on the argument **pcm**. The other loop has the same alias advice, plus a piece of alignment advice. The directions deal only with the alias advice, and leave the alignment advice for later.

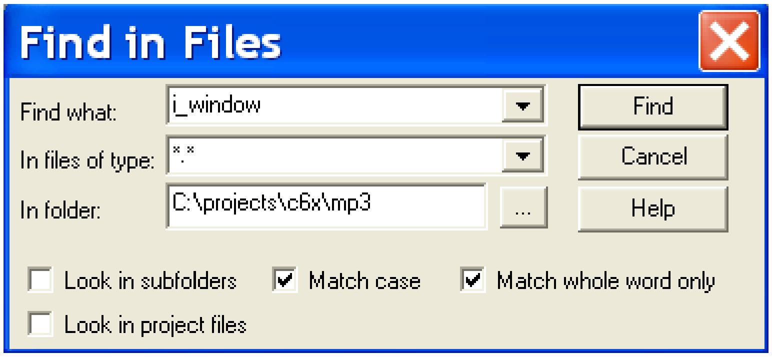
Tracing the origin of the argument **pcm** leads all the way back to the main call to the decoder in **towave.c**. At that point it is a pointer named **pcm_buffer**. Tracing that back shows that it is allocated some memory by a call to malloc. That cannot be an alias for anything else, so it is safe to restrict qualify **pcm** in i_window.

10. Add the restrict qualifier to **pcm**. The qualifier is placed between the asterisk and the name. That is because the qualifier applies to the pointer, and not the underlying memory. Type control-S to save the file.

```
void i_window(WININT * vbuf, int vb_ptr, short * restrict pcm)
```

11. Use the Find in Files feature to determine that the prototype for i_window is in the file isbt.c.

TIP: In the first Code Composer Studio toolbar row there is a group of four icons which all use binoculars. The one on the right is for Find in Files. Click on it to bring up the dialog. Conduct a search for i_window. See Figure 10 for the details. Results of the search appear in the Output window.



Find what: i_window

In files of type: *.*

In folder: C:\projects\c6x\mp3

□ Look in subfolders ☑ Match case ☑ Match whole word only

□ Look in project files

Figure 10. Find In Files for i_window

12. Restrict modify **pcm** in the prototype as well. Type control-S to save the file.

13. The next two rows are about different loops in the function back_bf. Explore the advice for each loop.

This situation is exactly the same as that in idx_forward_bf. The advice is to restrict qualify the array **f**.

14. Add the restrict qualifier to **f** for back_bf. Type control-S to save the file. There is no prototype of back_bf which needs to be updated.

```
static void back_bf(int m, int n, INT32 x[], INT32 f[restrict])
```

15. Several rows later there is a loop in the function back_bf0 with a piece of alias advice. Explore the alias advice. Leave the trip count advice for later.

This situation is exactly the same as the in idx_forward_bf. The advice is to restrict qualify the array **f**.

16. Add the restrict qualifier to **f** for back_bf0. Type control-S to save the file. There is no prototype of back_bf0 which needs to be updated.

```
static void back_bf0(int n, INT32 x[], WININT f[restrict])
```

Several changes have been made. It is a good time to build, run, and assess the resulting profile data.

CAUTION:
Any build or run causes the contents of the Profile Viewer to be discarded and updated with the new data.

To save the contents of the Profile Viewer, use the Save Current Data Set icon. Saved data can be viewed later with the Load Data Set icon. Details are in online help. Choose **Help | Contents | Application Code Tuning | Dashboard – Profile, Goals, Advice | Profile Viewer | Profile Viewer Toolbar**.

17. Build the application again. Under **Project** choose **Build** or click this icon .
18. Load the program. Under **File** choose **Reload Program**.
19. Profile the application again. Follow steps 4−13 of section 6.7.

    The Goals Window now shows a cycle count of approximately 952,000 cycles. This is a further reduction of 28%. Overall, the code runs 9X faster!

6.10 Implement Trip Count Advice

Following the trip count advice in this section reduces the cycle count another 12%. Overall, the code will run 10X faster!

These steps presume you have completed the profiling of the previous section.

1. If it isn’t open, launch the Profile Viewer. Click on the **Consultant** tab.
2. Sort the data in the column **cycle.CPU:Excl.Total**.

   There is a mix of alignment and trip count advice among the top loops. Focus on the trip count advice first. There is more trip count advice. And, section 4.5 advises addressing trip count advice over alignment advice.
3. The highest cycle count loop with trip count advice is in the function idx_forward_bf. Explore that trip count advice.

   The advice states the alternate loop would be removed if the compiler could guarantee the minimum trip count is at least four. The compiler can guarantee a minimum trip count of only two. The iteration count of the loop is given by n2. And n2 is assigned n >> 1. The value n is the second argument to back_bf.
4. Inspect all the calls to idx_forward_bf to determine the smallest value passed in for **n**.

   Note a #define maps idx_forward_bf to forward_bf, thus the search for calls to idx_forward_bf must actually look for forward_bf.

   TIP: Since idx_forward_bf is a static function, it is sufficient to search through the same file to find all the calls. However, using the Find in Files feature collects all the calls together in one place, which makes the inspection quick and easy.

   The second argument to idx_forward_bf is always passed a constant. The smallest constant passed is 4. Thus, the minimum trip count is 4 >> 1 > 2. Since that is less than 4, the advice to use the MUST_ITERATE pragma cannot be implemented.

   Further down in the Advice is an Alternate Suggestion. It says to use the –mh*n* option with a value of 16.

5. Click the help entry links to learn about the –mh option and the associated safety criteria for using it.

   In brief, the –mh*n* option tells the compiler it is legal to read n bytes before and/or after any array or block of data memory. Such speculative reads may enable the compiler to structure the software pipeline so that it can run at lower trip counts and thus eliminate the alternate loop.

   The simplest way to meet the requirements of –mh*n* is to insure there is some range of legal memory around all the compiler generated data sections. Those sections are named: .bss, .stack, .sysmem, .const. If you build with large data memory models, also include .far. If you use the DATA_SECTION pragma to create custom named data sections, include those as well.

6. Inspect the linker command file. In the Project View expand the tree to expose the file lnk.cmd. Double-click to open it.

   All the sections of interest are in the PMEM memory range. Because .text is at the beginning of PMEM, and PMEM is not full, all of these sections are firmly in the middle of PMEM. So, the padding requirements of –mh16 are well met.

7. Under the **Project** menu, choose **Build Options** …

8. In the **Build Options** dialog, click the **Compiler** tab.

9. Click on the **Advanced** item in the **Category** list.

10. Enter 16 in the box for **Speculate Threshold (−mh).**

11. Click **OK** to close the **Build Options** dialog.

    Now –mh16 is part of the build options.

12. The next highest cycle count loop with trip count advice is from the function back_bf0. Note there are two loops from back_bf0 with trip count advice. Explore the advice for both loops.

    The advice for the loop with the higher cycle count states the alternate loop could be removed if the compiler knew the minimum trip count was at least 3. The compiler can guarantee a minimum trip count of only 1. The advice is to either use a MUST_ITERATE pragma to indicate the minimum trip count is 3, or use the option –mh16.

    The advice for the other loop states the alternate loop could be removed if the compiler knew the minimum trip count was at least 6. The compiler can guarantee a minimum trip count of only 4. The advice is to either use a MUST_ITERATE pragma to indicate the minimum trip count is 6, or use the option –mh24.

    The iteration count of both loops depends on the argument **n**. Inspection of all the calls to back_bf0 shows that a constant is always passed to **n**. The smallest constant passed is 8. It is left as an exercise to the reader to show that each loop executes **n**/2 times. So, the actual minimum trip count of each loop is 4.

    Table 1 summarizes the key information about the two loops.

Table 1. Trip Count Information

Loop	Actual TC	MUST_ITERATE	−mh*n*
#1	4	3	16
#2	4	6	24

NOTE: Loop = which loop
Actual TC = Actual minimum trip count when the loop executes
MUST_ITERATE = The value for the MUST_ITERATE pragma recommended in the advice
−mh*n* = The value for the speculative load option recommended in the advice

With an actual minimum trip count of 4, one loop could use the recommended MUST_ITERATE pragma, while the other could not. The loop that cannot use the MUST_ITERATE pragma recommends, as an alternative, using the option –mh24. Note that –mh24 is a higher speculative read threshold than the –mh16 recommended for the other loop, or the –mh16 already applied for the loop in the function idx_forward_bf. So, changing from –mh16 to –mh24 addresses all of the advice recommended so far.

13. Redo steps 7−11, but with a –mh*n* value of 24, instead of 16.
14. The next highest cycle count loop with trip count advice is back_bf. Note that, like back_bf0, there are two loops that have trip count advice. Explore the advice for both those loops.

    This situation is very similar to that of back_bf0. The conclusion is to use the speculative load threshold –mh24, which is already in place.
15. Build the application again. Under **Project** choose **Build** or click this icon .
16. Load the program. Under **File** choose **Reload Program**.
17. Profile the application again. Follow steps 4−13 of section 6.7 Sort the Loops.

    The Goals Window now shows a cycle count of approximately 836,000. This is a further reduction of 12%. Overall, the code runs over 10 times faster!

6.11 Implement Alignment Advice

Following the advice in this section reduces the cycle count another 0.5%. Even so, alignment advice might occur in a key inner loop in your application, and knowing the techniques applied in this section could prove invaluable.

The directions in this section presume use of the profile data collected in the previous section.

1. If it isn’t open launch the Profile Viewer. Click on the **Consultant** tab.
2. Sort the data on the column **cycle.CPU:Excl. Total**.

   The highest ranking loops with advice have Alignment Advice.
3. Double-click the Advice List cell for the loop in i_window. Explore the advice. Especially look at the online help entry on methods to align data.

   The advice indicates that, when the compiler knows **pcm** is aligned, it may use instructions that access more than one short in a single instruction. These are the SIMD instructions discussed in section 4.4. The alignment is required by the SIMD instructions. The suggestions states addressing three items. The next three steps address these items in turn.

Recall that in section 6.9 step 11, the origin of the argument **pcm** to the function i_window is traced back to the variable **pcm_buffer** in **towave.c**. The actual expression passed is **pcm_buffer + pcm_bufbytes**. The variable **pcm_buffer** is allocated some memory by a call to malloc. The online help entry on aligning data indicates pointers allocated by malloc() are always aligned. Now consider the pointer expression. Tracing the value of **pcm_bufbytes** shows that it is always 0 when the call occurs, and thus has no effect on the alignment of the expression. Perhaps an earlier revision of the code may have computed a non-zero value into **pcm_bufbytes**.

4. Because **pcm_buffer** and the expression **pcm_buffer + pcm_bufbytes** are both known to be aligned, no explicit alignment steps are necessary.
5. To tell the compiler that **pcm** is aligned, cut and paste the _nassert macro from the Advice into the source of the i_window function. Place it near the top of the function, after the variable declarations. Save the file with control-S.
6. Since the bounds of the loop are constant, there is no need to indicate the loop unroll factor with a MUST_ITERATE pragma.
7. Bring up the alignment advice for the function i_dct32.

   The advice states the compiler can generate better code if it knows that **x** is aligned. Tracing the origin of **x** shows that is the static array **samples** in the file **iup.c**. The online help entry on aligning data states such arrays are already aligned by the compiler, and further gives an example macro named ALIGNED_ARRAY for telling the compiler about such alignment.
8. Because **itype.h** is included by most of the files, place the ALIGNED_ARRAY macro at the end of that file. Copy it from the online help entry. Save the file with control-S.
9. Apply the macro to the argument **x**. Place it near the top of the function, after the variable declarations. Save the file with control-S.

   ```
   ALIGNED_ARRAY(x);
   ```
10. Build the application again. Under **Project** choose **Build**.
11. Load the program. Under **File** choose **Reload Program**.
12. Profile the application again. Follow steps 4−13 of section 6.7.

    Now the Goals Window shows a cycle count of approximately 831,000 cycles, a further reduction of 0.5%.

6.12 Implement the Last Round of Advice

The advice is this section is mostly applied to functions that do not execute in the present configuration of the MP3 decoder. Full testing of the decoder would use all the configurations, and thus exercise all the code. Therefore, the changes described here could become important. This last round of advice is implemented in the mp3_tuned project which accompanies this application note.

A loop in the function i_window still has some alignment advice. This time it states the desired alignment is 8 bytes, not 4. This is a case where following previous advice changes the optimization opportunities in the code such that further advice is generated. Implement this additional advice by following step 5 of the previous section, but change the alignment factor from 4 to 8.

A loop in the function i_dct8 has some alignment advice. The situation here is the same as that for the function i_dct32. Follow step 9 of the previous section.

Collect all the advice together by type by sorting on the Advice List column. Notice how most of the alias advice is for functions with a name that begins with i_window. Some examples are i_window8 and i_window16. For the sake of this discussion, the term **i_window-like** refers to all of these functions. Recall the discussion about alias advice in section 6.9 with regard to the function i_window. See step 9. Inspecting how all the i_window-like functions are called shows that the **pcm** argument to those functions can also be traced back to **pcm_buffer** in towave.c. So, for all those functions it is safe to restrict qualify the pcm argument. Perform steps 10 and 12 of section 6.9 on all the i_window-like functions.

Four of the i_window-like functions also have some alignment advice that is similar to the alignment advice for i_window discussed in section 6.11 step 3. In similar fashion, add the _nassert pragma to indicate that **pcm** is 8-byte aligned.

Only inline advice remains. The term inline refers to inlining a function call. A function call is inlined when, instead of calling the function, the entire body of the function replaces the call. As such, inlining is typically applied to small functions. Any loop which contains a call must inline that call to be optimized further. None of the functions in this example which might be inlined are small enough. Therefore, none of the inline advice can be implemented.

6.13 Performance Summary

The options advice makes the code run 6.4 times faster. The rest of the Consultant advice, combined with the options advice, makes the code run 10.3 times faster. The performance improvement due to the remaining types of advice is shown in Table 2.

Table 2. Performance Improvement by Advice Type

Advice	Incremental	Overall
Alias	28.5%	28.5%
Trip Count	12.2%	37.2%
Alignment	0.5%	37.5%

This table shows the further performance improvement gained after implementing the options advice. The incremental improvement column shows the improvement due to implementing that advice alone. The overall improvement column shows the improvement due to implementing that advice in combination with all the previous advice.

7 Summary

Compiler Consultant is a powerful ally in the code tuning process.

The performance improvement on the MP3 example speaks for itself: 10.3 times faster. About 60% of that performance lift comes from enabling optimization. The remaining 40%, however, comes from implementing the more sophisticated advice.

All this performance improvement comes quickly and easily. Compiler Consultant finds the exact points where optimization opportunities exist. These opportunities can be rank ordered several different ways. Detailed explanations illustrate how to change the code. The developer, in the main remaining role, insures the proposed changes are safe, and then implements them. All this improvement is typically achieved in a few hours.

Consultant presents the advanced features of TI compilers in context of optimization opportunities in actual code. As such, the power of these features presents more clearly than in any documentation.

In conclusion, Compiler Consultant combines significant performance improvement and reduced development time into a very compelling package.

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
		Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments
Post Office Box 655303 Dallas, Texas 75265

Copyright © 2004, Texas Instruments Incorporated