

Using CacheTune (Code Composer Studio v3.0) to Improve Cache Utilization on TMS320C6000 Targets

Ning Kang

Software Development Systems

ABSTRACT

Many of today's digital signal processors (DSPs) have incorporated cache into the on-chip memory to support higher clock rates. While cache improves processor throughput by reducing the average memory access time, sub-optimal cache usage causes some performance overhead and could become a critical bottleneck in the system. Maximizing cache effectiveness becomes a key to boosting the overall system performance.

CacheTune is a new tool that helps the developer attain high levels of cache efficiency by addressing the issues in cache visualization, analysis and optimization. It graphically visualizes program and data cache accesses over time, which enables quick and effective reorganization of non-optimal cache utilization. The tool also provides proactive advice in guiding the developer to analyze the memory accesses patterns and tune the cache memory subsystem to meet performance goals.

This application report introduces the CacheTune tool, discusses the recommended code development flow to tune your application for a high level of cache efficiency and uses an example to illustrate the necessary steps required to utilize the CacheTune tool.

Contents

| | | |
|-------------------|--|-----------|
| 1 | Introduction | 3 |
| 2 | CacheTune Overview | 3 |
| 2.1 | Development Flow to Increase Cache Efficiency | 6 |
| 3 | Using CacheTune with an Example | 8 |
| 3.1 | Application Validation | 9 |
| 3.2 | Data Collection | 12 |
| 3.3 | Data Cache Visualization and Optimization | 17 |
| 3.3.1 | Applying Application Level Optimization: EDMA Double Buffering Framework | 23 |
| 3.3.2 | Applying Procedural Level Optimization: Restructuring the Data Layout | 25 |
| 3.3.3 | Exploiting Miss Pipelining | 28 |
| 3.4 | Program Cache Visualization and Optimization | 29 |
| 3.5 | Overall System Improvement | 33 |
| 4 | Conclusion | 34 |
| 5 | References | 34 |
| Appendix A | Cache Basics | 35 |
| Appendix B | Cache Structure on TI C6000 DSPs | 37 |
| B.1 | TMS320C6000 Two-Level Cache | 37 |

Trademarks are the property of their respective owners.

List of Figures

| | | |
|-----------|---|----|
| Figure 1 | A First Look at CacheTune (Data Cache View) | 4 |
| Figure 2 | CacheTune General Advice | 5 |
| Figure 3 | Advice for Program Cache | 6 |
| Figure 4 | Development Flow to Increase Cache Efficiency | 7 |
| Figure 5 | Code for Processing Chain | 9 |
| Figure 6 | Linker Command File | 10 |
| Figure 7 | Graph Property Dialog Window | 11 |
| Figure 8 | Input and Output Images Data Collection | 12 |
| Figure 9 | Range Tab of the Profile Setup Window | 13 |
| Figure 10 | Add Control Point Dialog Window | 14 |
| Figure 11 | Goals Window | 15 |
| Figure 12 | Trace Information Window | 17 |
| Figure 13 | Overview of Data Cache Accesses | 18 |
| Figure 14 | Sequential Accesses to Data Buffers | 19 |
| Figure 15 | Access Patterns to Data Buffers | 20 |
| Figure 16 | Cross Cache | 22 |
| Figure 17 | EDMA Double Buffering Framework | 23 |
| Figure 18 | Data Cache Accesses After Using EDMA | 26 |
| Figure 19 | Data Cache Accesses After Restructuring | 28 |
| Figure 20 | Viewing the Symbol | 30 |
| Figure 21 | Conflict Misses in Program Cache | 31 |
| Figure 22 | Modified Linker Command File | 32 |
| Figure 23 | Program Cache After Optimization | 33 |

List of Tables

| | | |
|---------|---------------------------------------|----|
| Table 1 | Development Flow | 7 |
| Table 2 | Profile Date of Initial Run | 15 |
| Table 3 | Profile Data Comparison 1 | 25 |
| Table 4 | Profile Data Comparison 2 | 27 |
| Table 5 | Profile Data Comparison 3 | 29 |
| Table 6 | Profile Data Comparison 4 | 32 |
| Table 7 | Overall Cache Events Comparison | 34 |

1 Introduction

The TMS320C621x™, TMS320C671x™, and TMS320C64x™ digital signal processors (DSPs) of the TMS320C6000™ DSP family employ a highly efficient two-level cache-based memory architecture for on-chip and for external memory accesses program and data accesses. Cache reduces the average memory access time by exploiting the locality of the memory accesses, hence improving the CPU throughput. Using the cache subsystem effectively allows the DSP developer to meet real-time requirements.

The CacheTune tool, available in Code Composer Studio™ v3.0, is aimed at identifying inefficient cache usage and providing proactive advice to facilitate making rapid improvements in cache performance. By using color-coding schemes for different accesses in the cache, the CacheTune tool provides a way to visualize the memory reference by address range over time, at different resolutions of time. It effectively helps the developer to recognize inefficient cache usage. Furthermore, the tool provides proactive advice in assisting the developer to analyze the graphical display and identify the memory accesses patterns. Cache optimization techniques and code examples are also provided that describe how to tune the application to improve cache performance. Using this tool, the developer can quickly improve cache usage and thereby optimize the CPU cycles consumed in the cache subsystem.

2 CacheTune Overview

This section introduces the CacheTune tool and describes the recommended code development flow to improve cache efficiency.

The CacheTune tool utilizes software simulation platforms. All the C621x, C671x and C64x device and functional simulators are capable of collecting cache data that can be visualized by CacheTune.

The profiled memory reference data can be viewed inside a two-dimensional display grid. The accessed addresses are plotted along the vertical axis while access times are plotted along the horizontal axis. All the accesses are color-coded by type, for example, cache hits are shown as green pixels while cache misses are shown in red. The tool can also display symbolic information when available, such as symbol name, size and section, etc., which allows for easy identification of functions and major data structures. In addition, the tool provides various filters, panning and zoom features to navigate and drill down into inefficient cache memory areas. This visual temporal and spatial view of cache accesses enables quick identification of sub-optimal cache usage areas. Issues such as functions conflicting with one another and data accesses that make insufficient use of the cache can be easily discovered.

For C6000 devices, CacheTune supports three graphical views: program cache, data cache and cross cache. Each view tracks different aspects of the cache memory system behavior. They can be viewed separately by clicking on the cache tabs under the toolbar. Each type of the graphical view and the supported events are described below. For certain cache terminologies, refer to related Code Composer Studio online help topics.

- Program cache

This correlates program memory references with the outcomes in level-one program cache (L1P) and level-two (L2) unified memory. The supported events are cache hit and cache miss.

- Data cache

This correlates data memory references with the outcomes in level-one data cache (L1D) and level-two (L2) unified memory. The supported events are memory read, memory write, cache hit and cache miss.

- Cross cache

This encodes a cross-reference of instruction memory references with the cache outcome of level-one data cache. The addresses are the same as for the program cache display, but the events displayed are with respect to the data memory references contained within those instructions. The supported events are memory read, memory write, cache hit and cache miss.

Figure 1 demonstrates a sample CacheTune window and some annotations.

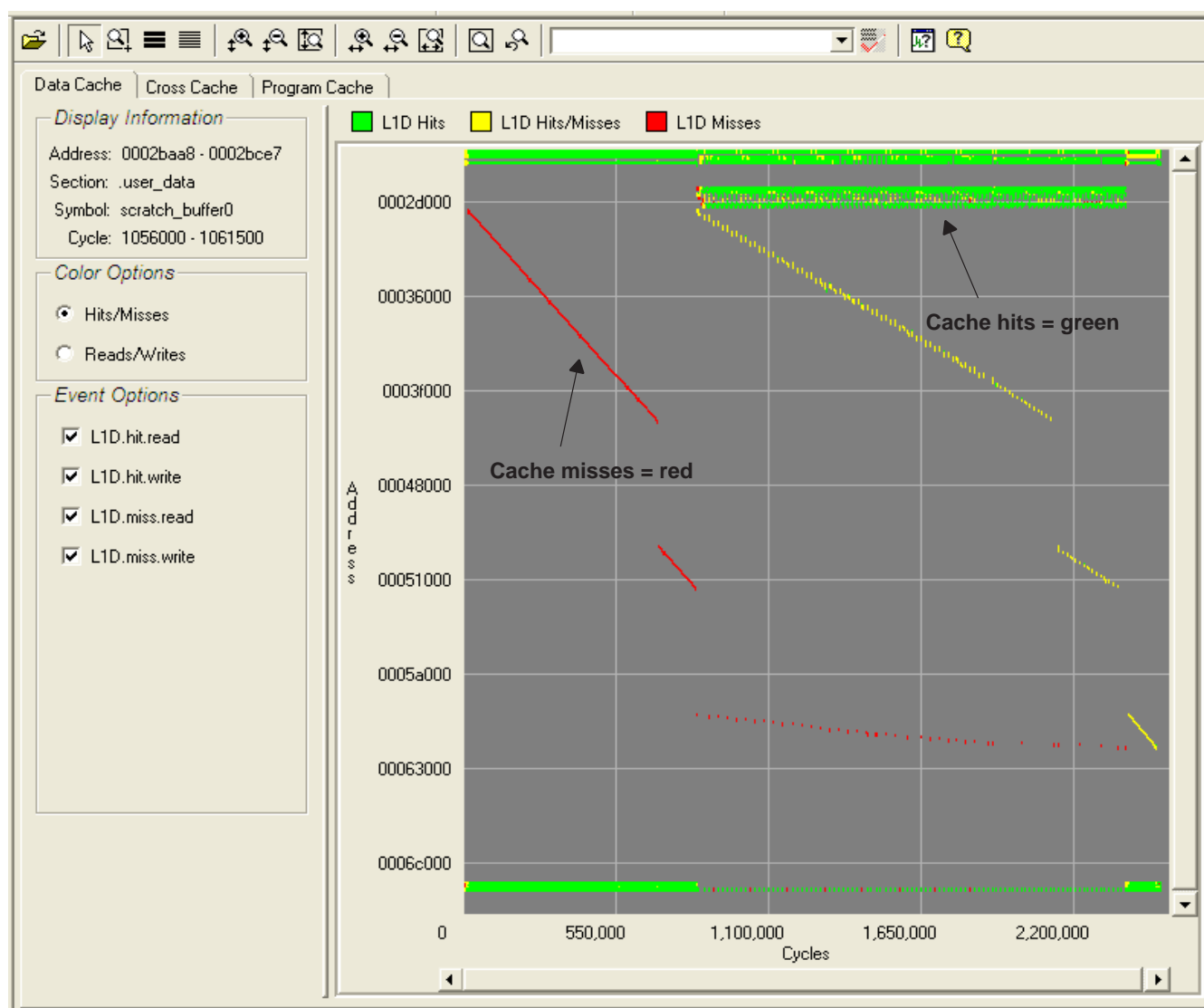


Figure 1. A First Look at CacheTune (Data Cache View)

One key feature that CacheTune offers is the proactive advice that helps you utilize the CacheTune tool to increase cache efficiency. The advice is displayed in an Advice Window, which is automatically displayed after launching the tool. There are cache-specific advice topics available for each graphical view as well as a general advice topic. The general advice is the first advice displayed. It tells you how to collect cache data and what steps to take next. A screen shot of the CacheTune general advice window is shown in Figure 2.

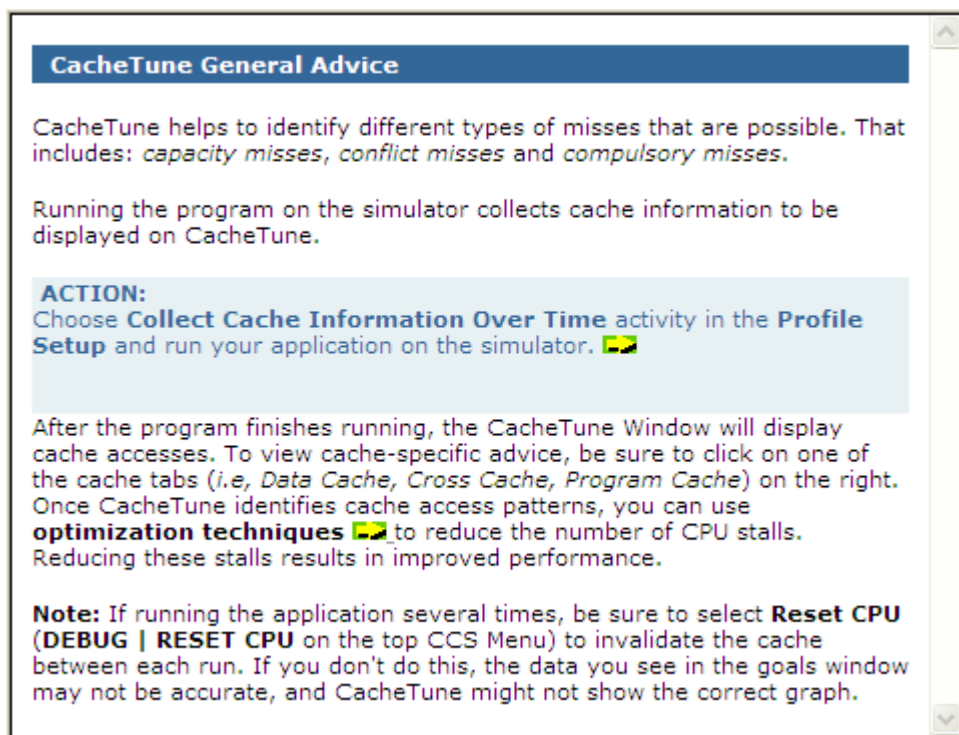


Figure 2. CacheTune General Advice

The cache-specific advice tells you what type of misses are possible in the data and program caches, why these misses occur, what pattern to look for in the CacheTune graphical display and what steps you can perform to reduce those misses. The cache misses in data and program caches are categorized into different miss scenarios. The cause of each miss scenario and possible optimization techniques are described. There are links to online help topics that show how each type of misses appears in the CacheTune graph. This can help developers interpret the graphical display and identify the miss patterns of their application. Once those misses are identified, corresponding optimization techniques can be applied to reduce the misses. Examples are provided to illustrate how to modify your program to employ certain techniques. Figure 3 shows a screen shot of program cache advice window.

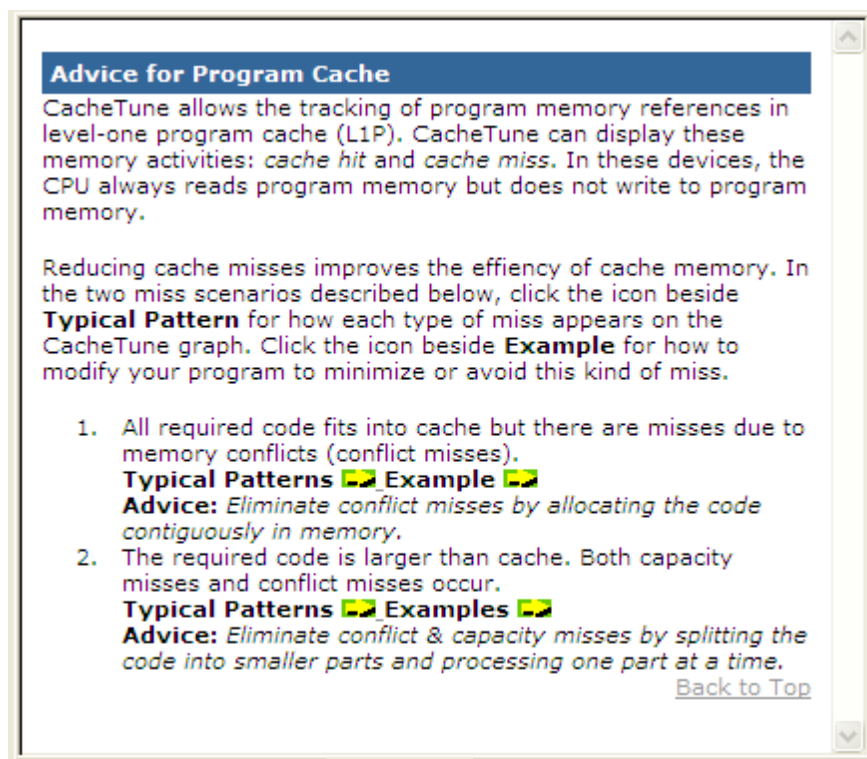


Figure 3. Advice for Program Cache

To better use the CacheTune tool, it is beneficial to understand the fundamental concepts of cache and the characteristics of the cache memory architecture. These are discussed in Appendix A and B respectively. The CacheTune tool can be invoked from the Profile → Tuning menu in Code Composer Studio. Section 3 discusses the procedures to set up and invoke the tool in detail with a code example.

2.1 Development Flow to Increase Cache Efficiency

The recommended code development flow involves utilizing the CacheTune tool to aid your optimization to increase cache performance. The flow consists of three phases that are illustrated in Figure 4.

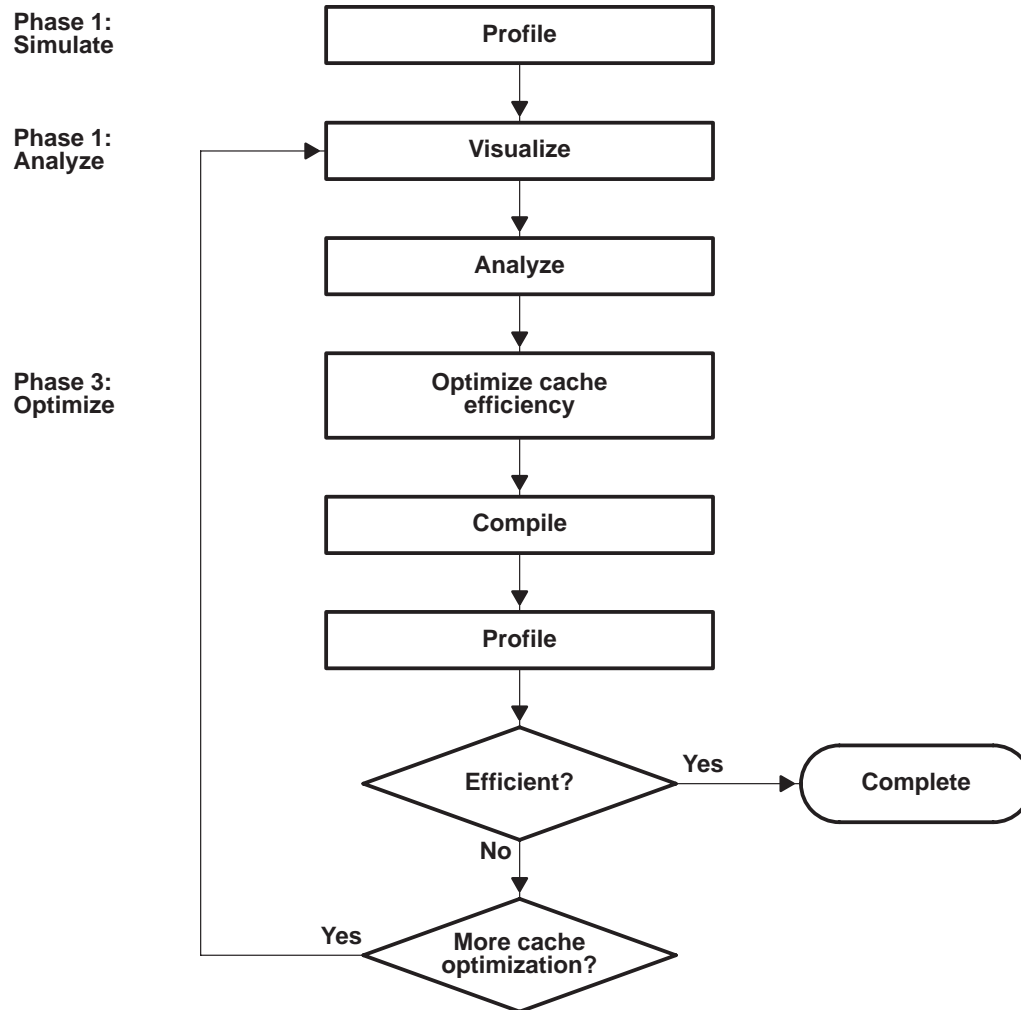


Figure 4. Development Flow to Increase Cache Efficiency

The goals for the phases in the development flow are described in Table 1.

Table 1. Development Flow

| Phase | Goal |
|-------|--|
| 1 | Run the program to collect cache information and use the profile viewer tool to count the occurrence of cache events, such as the number of cache misses and CPU stalls caused by cache misses, etc. To analyze the cache behavior of your code, proceed to phase 2. |
| 2 | Visualize the memory reference patterns in the CacheTune tool to identify the areas of code and data that are incurring cache misses. Use the advice to identify the miss patterns. To improve the efficiency of cache, proceed to phase 3. |
| 3 | Apply optimization techniques and transformations to improve cache efficiency. Use the profile viewer tool to check the improvement. If the code is still not as efficient as you would like, repeat steps in phase 2 and 3 until you are satisfied. |

All three phases can be achieved within the simulator environment. The device simulator is required to accurately measure the total execution cycles as well as the cache stall cycles of the application.

The typical user workflow reflecting the CacheTune usage in each development phase is described:

1. Compile and profile the application with device simulator
 - Use Code Composer Studio setup to select the device simulator
 - Use Profile Setup tool to instruct the simulator to collect cache information
 - Run the application to collect cache data and view the profile data from Profile Viewer
2. Visualize and analyze the memory accesses from CacheTune
 - Visualize the cache events with CacheTune
 - Locate the areas of caches misses
 - Use the typical miss patterns from cache-specific advice to analyze the display and identify the miss scenario, that is, the classes of misses
3. Optimize cache performance
 - Apply the recommended optimization technique for each miss scenario using the provided example for references

Steps 1–3 can be repeated as needed until the cache stall cycles are reduced to meet particular efficiency needs.

3 Using CacheTune with an Example

This section walks you through the code development flow to increase the cache performance. An image processing code example is provided here to illustrate how to use the software development tools in each phase of the development flow. The example consists of a processing chain that processes an input image, utilizing routines from the Texas Instruments C64x Image and Video Processing Library (IMGLIB). The example uses C64x device as a target, but also conceptually applies to C621/C671x devices.

The complete source code is provided with this documentation. In order to get the most out of the examples, it is recommended that you work through the instructions in the text. To do so, install accompanying self-extracting file into directory C:\CCStudio\myprojects. If Code Composer Studio is not installed in the default directory, you need to change the path accordingly.

The following sequence illustrates the steps needed with each phase of the development flow. It is based on the assumption that you are already familiar with Code Composer Studio IDE (know how to set up Code Composer Studio as well as to create and build a project in Code Composer Studio). Some simplifications are used to keep the content manageable in length. Refer to *TMS320C6000 DSP Cache User's Guide* (SPRU656) for detailed descriptions on the features and discussion on cache optimization techniques.

3.1 Application Validation

Step A: Setting up the device simulator

The first step is to select the C6416 device simulator, which provides cycle accurate program execution time. Cycle accurate CPU execute time is necessary to measure the overall application performance. We will assume you are familiar with the Code Composer Studio setup interface.

1. Start the Code Composer Studio setup utility.
2. From the list of available configurations within the Import Configuration dialog box, select the standard configuration C6416 Device Cycle Accurate Simulator, Little Endian as the target simulator.
3. Save the configuration and exit the setup utility.

Step B: Opening and examining the project

1. Start Code Composer Studio.
2. Choose the Project → Open menu item. Open the wave_horz.pjt project in the directory c:\CCStudio\myprojects\wave_hoz\original.
3. Expand the Project View list by clicking the + signs next to Projects, wave_horz.pjt, and Source. The files used in this program include:
 - main.c: This file contains main function that utilizes APIs from TMS320C6000 Chip Support Library (CSL) to configure 256K bytes of level-two (L2) memory as cache and calls the function proc_chain. For more information on the CSL functions, refer to *TMS320C6000 Chip Support Library API Reference Guide* (SPRU401).
 - proc_chain.c: This file contains function proc_chain and related data declaration. The function proc_chain calls the IMGLIB functions to process the image data, as shown in Figure 5.
 - Ink.cmd: This is the linker command file for this project.
4. Double-click on each program file to open it. Examine the source code for this program in the editor window of Code Composer Studio. Notice the following aspects of the program:

In this example, a horizontal wavelet filter (IMG_wave_horz) is applied to an image that is located in external memory. Since the filter routine operates on 16-bit data, two additional routines (IMG_pix_expand and IMG_pix_sat) are required, which convert the 8-bit image data to 16-bit data, and vice versa. All three routines are taken from C64x IMGLIB. For more information on the three functions, refer to *TMS320C64x Image/Video Processing Library Programmer's Reference* (SPRU023).

```
IMG_pix_expand(IMG_SIZE, in_image_ext, expand_out);
for (j=0; j<(IMG_SIZE/COLS); j++)
    IMG_wave_horz (&expand_out[j*COLS], qmf, mqmf, &wave_out[j*COLS], COLS);
IMG_pix_sat(IMG_SIZE, wave_out, out_image_ext);
```

Figure 5. Code for Processing Chain

Since the data buffers are too large to fit into the level-two (L2) memory, they are allocated in external memory, which is identified as CE0 in this case. All other sections are allocated into L2 memory configured as RAM while 256 Kbytes of level-two memory are configured as cache. Furthermore, by relative placement in the linker command file, the three functions: IMG_pix_expand, IMG_pix_sat and IMG_wave_horz are placed into memory so that they are 0x4000 (16 K) bytes away from each other. For more information on the linker command file, refer to *TMS320C6000 Assembly Language Tools User's Guide* (SPRU186). Part of the linker command file is listed in Figure 6.

```
MEMORY
{
    VEC:          o = 00000000h    l = 00000020h
    SRAMT         o = 00000020h    l = 00011FE0h
    SRAM:         o = 00020000h    l = 000A0000h
    CE0:          o = 80000000h    l = 01000000h
}

SECTIONS
{
    . . .
    GROUP          >    SRAMT
    {
        .text
        /* Functions are placed 16K bytes away from each other by inserting holes */
        .text:_pix_expand
        .text:_pix_sat {
            . += 0x3F80; /* Create a hole with size 0x3F80 */
        }
        .text:_wave_horz {
            . += 0x3F80; /* Create a hole with size 0x3F80 */
        }
    } /* End of GROUP */
    . . .
}
```

Figure 6. Linker Command File

Step C: Validating the output

1. Compile and load the program.
2. Run the program to completion.

Code Composer Studio provides a graph menu which contains many options in displaying your image data. You can display a graph to validate the image-processing algorithm.

3. From the menu, select View → Graph → Image. Modify the Dialog as shown in Figure 7 and click OK. This displays the original input image in the Graphical Display window.

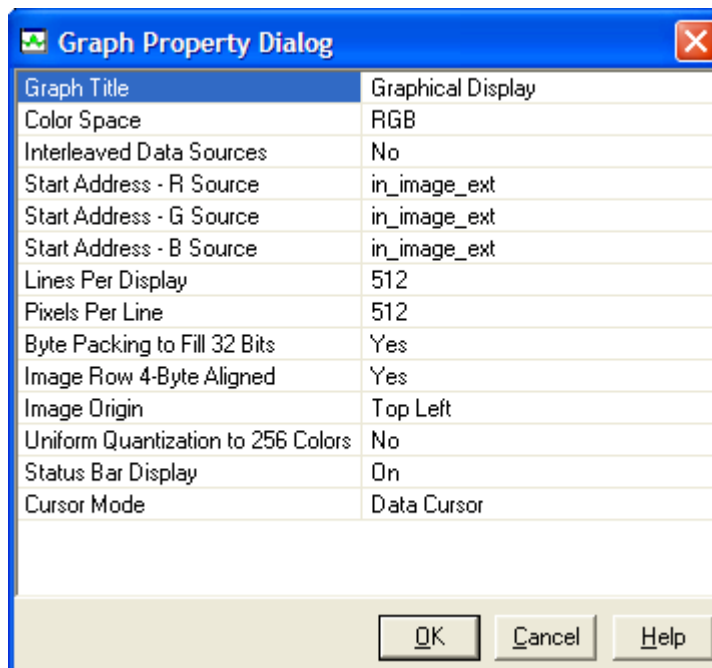


Figure 7. Graph Property Dialog Window

- Repeat step 3 to display the output image in another Graphical Display window by replacing "in_image_ext" with "out_image_ext". The input and output images are shown in Figure 8.

You have successfully built your project and validated the outputs through the Graphical Display window. These are achieved under the standard layout, also known as the debug layout, which enables you to build and debug your code. You are now ready for the tuning phase aimed at increasing the efficiency of your application. In addition to the standard (debug) layout, Code Composer Studio provides a Tuning layout that is meant to focus your attention on the optimization needs. As the example is using already optimized assembly routine from IMGLIB, you will be mainly focusing on minimizing the cache overhead.

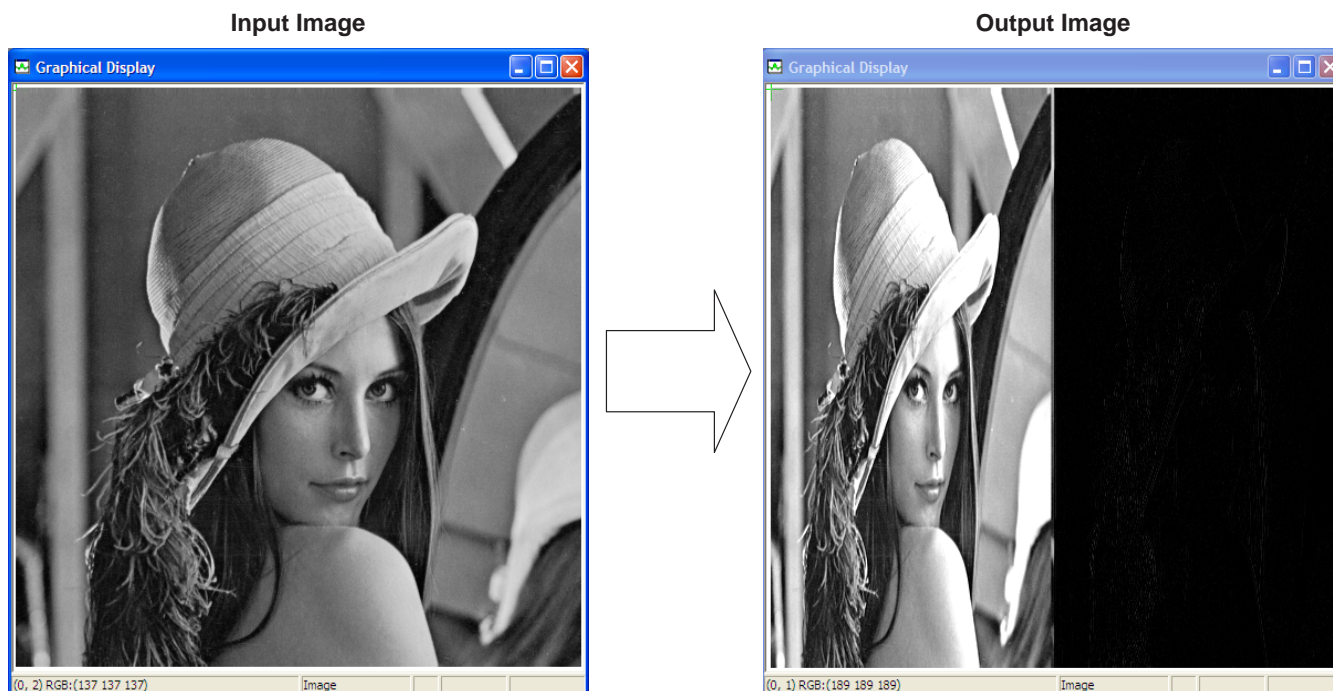



Figure 8. Input and Output Images Data Collection


3.2 Data Collection

Step A: Switching to the tuning mode

1. Activate tuning mode by clicking the tuning fork icon or by clicking the menu item View → Layout → Tuning Layout. Switching to Tuning mode reorganizes the workspace to provide a good view of both the Project and the Advice windows, which helps guide the tuning process. If you are the first-time user, it is highly recommended to follow the advice within the advice window.
2. From the Welcome to Tuning advice window, click the Setup Advice icon  under the action section. Examine the advice on Profile Setup.
As prompted from the advice, you need to perform a CPU reset then reload the program before data collection. This is to ensure accurate profile results.
3. To reset CPU, either click the link in the Setup advice window or select Reset CPU from the Debug menu.
4. From File menu, select Reload the Program.

Step B: Selecting the type of profile data

In this step, you will instruct the simulator to collect the type of profile data for cache tuning using the Profile Setup tool. For more information on Profile Setup, refer to Code Composer Studio on-line help.

1. Within the Setup advice window, click the Profile Setup icon . This opens the Profile Setup tool on the right side of the window. You will use the Profile Setup tool to instruct the simulator to profile your application.
2. From the Profile Setup toolbar, toggle on the Enable/Disable Profiling button. This enables the profiling capacities in Code Composer Studio.
3. By default, Profile Setup displays the Activity tab. Select the activities Collect application level profile for total cycles and code size. Collect data on Cache Accesses over a specific address range and Collect cache Information over time. Clicking each activity displays a detailed description in the lower portion of the Activity tab. These activities collect the most common types of data that are necessary for cache tuning.
4. Other events needed for tuning cache performance for this application can be manually selected from the Custom tab. Click the Custom tab of the Profile Setup window. Observe that some CPU, L1D, and L1P events have already been selected as a result of selecting the activities in the Activity tab from the previous step. Select the following events from the range pane (top portion of the custom tab):
 - L2.cache.miss.data.read
 - L2.cache.miss.data.write
 - L1D.stall.write_buf_full
 - cycle.total
 - cycle.CPU
5. From time pane (lower portion of the custom tab), select the following events:
 - data.L2.cache.hit.data.read
 - data.L2.cache.hit.data.write
 - data.L2.cache.miss.data.read
 - data.L2.cache.miss.data.write

A screen shot of the range tab with the events selects is shown in Figure 9.

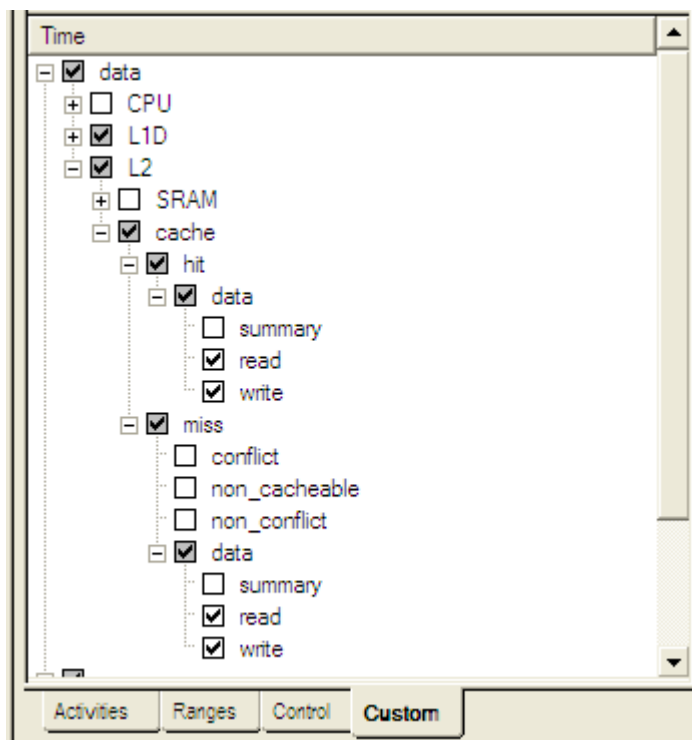


Figure 9. Range Tab of the Profile Setup Window

Step C: Selecting the range of profile data

In this step, the range of profile data will be selected. You will create Halt Collection and Resume Collection points (control points) to exclude the initialization area of code.

NOTE: In order to create control points at the desired position, the source file `main.c` is compiled with file specific compiler option: `-g` and no optimization. As file `main.c` only contains set up code, the specific option has least or no effect on the performance. For more information on using file specific compile options, see Code Composer Studio online help.

1. Switch to Control tab of Profile Setup.
2. Right click on the Halt/Resume collection pane (lower pane) and select create halt collection point... from the context menu. This brings up the Add Control Point dialog box.
3. By default the Symbol ratio button is selected. Enter `c_int00` in the Function Name field, as shown in Figure 10. `C_int00` is the program's entry point and creating a halt collection point here will stop the data collection at the beginning of the application.

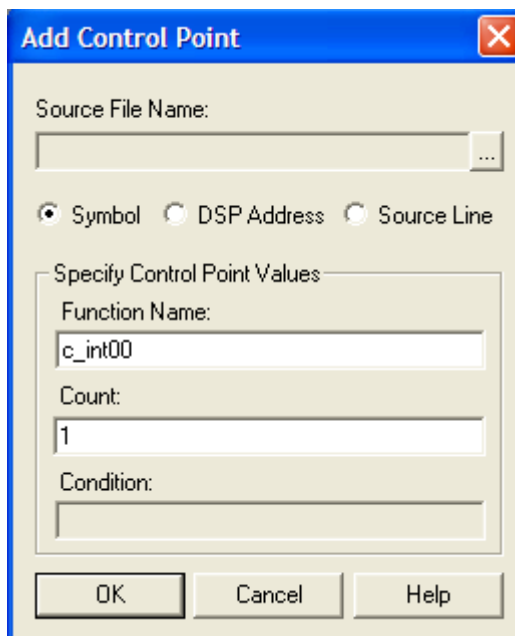
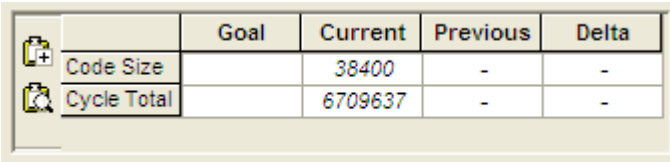


Figure 10. Add Control Point Dialog Window

4. From file main.c, highlight the line of calling function proc_chain and drag-and-drop it to the Resume Collection pane within the Control tab. Data collection will now resume when the proc_chain function is reached.
5. From the same file, highlight the line of the closing brace (}) of the main function and drag-and-drop it to the Halt Collection pane in the Control tab.
The added Halt Collection and Resume Collection points are displayed in the Halt/Resume Collection Pane of Profile Setup window. By setting up the control points this way, you will only profile the key functions and exclude any other initialization and setup routines.
6. Select Enable/Disable All Functions icon from the toolbar. Switch to Range tab and you will notice the program's functions appear within the Function branch.
7. Hide the Profile Setup window by right-clicking on the window and selecting Hide from the context menu. Closing un-used window is a good practice to regain window space. Profile setup can be re-launched easily when needed and all the selected events will be retained.

Step D: Viewing profile data

1. Run the application to completion.
2. Invoke Goals Window by selecting menu item Profile → Tuning → Goals. Goals Window displays total cycles and code size at the application level. You will focus on the total cycle count to track the improvement on the performance at the application level. Figure 11 is a screen shot of this window.



| | Goal | Current | Previous | Delta |
|-------------|------|---------|----------|-------|
| Code Size | | 38400 | - | - |
| Cycle Total | | 6709637 | - | - |

Figure 11. Goals Window

- From Profile menu, select Viewer to open the Profile Viewer window. Profile Viewer displays the selected events after the program finishes. In the Profile Viewer window, look at the number in the inclusive total columns of the proc_chain function. The Inclusive Total column shows the total number of cycles spent executing the profile area, including the execution time (cycle count) of any subroutines called from within the profile area. So the inclusive total events of proc_chain function also include all subroutines called by this function. The profile results are summarized in Table 2.


Table 2. Profile Data of Initial Run

| Profile Events | Cycles | Profile Events | Counts |
|--------------------------|-----------|--------------------------|---------|
| CPU.stall.mem.L1D | 6,068,428 | L1D.misses.summary | 380,931 |
| CPU.stall.mem.L1P | 132 | L1P.misses.summary | 29 |
| Cycle.CPU | 641,062 | L2.cache.miss.data.read | 10,240 |
| Cycle.Total | 6,709,606 | L2.cache.miss.data.write | 10,240 |
| L1D.stall.write_buf_full | 2,722,684 | Cache overhead | 947% |

The cache overhead can be assessed by measuring the pure CPU executed cycles and cache stall cycles: $(\text{CPU.stall.mem.L1D} + \text{CPU.stall.mem.L1P}) / \text{Cycle.CPU}$. For more detailed discussion on cache overhead, refer to the related CacheTune online help topic. The cache overhead now is roughly 947% $((6,068,428 + 132) / 641,062)$. Most of the cache overhead is from data cache stalls that are caused by misses when accessing data from L1D and L2 cache as well as write buffer full occurrences. Your numbers may vary.

Step D: Saving the profile data set

Rather than manually recording the profile results for each execution, you can save the profile data set from Profile Viewer and restore them later for comparison.

- Click the Save Current Data Set icon  in the Profile Viewer window.
- After the Save As window appears, save the current data set as run0.xml in a new folder named Dataset.

So far, you have completed phase 1 of the development discussed in section 2. Knowing the cache performance is the bottleneck of the application, you are ready to proceed to the next phase: visualize and analyze the cache accesses. Your numbers may vary.


3.3 Data Cache Visualization and Optimization

Step A: Visualizing and analyzing data cache

To achieve high cache utilization, you need to have a thorough understanding of where and when memory accesses occur and what kind of accesses, that is, hits or misses, writes or reads. You also need to know which data accesses by which function demonstrates non-optimal cache usage. This information can be obtained by visualizing and analyzing the CacheTune graphical display.

1. Select the menu item Profile → Tuning → CacheTune. This launches the CacheTune tool in the main editor window and also activates the CacheTune tab in Advice Window. By default, the CacheTune output graph window shows the cache accesses for data cache.

NOTE: Always start with data cache because the code may be modified after you optimize for data cache performance, thus changing the memory access pattern of the program cache.

2. Click the Show Trace Information button  from the toolbar to display the trace information. This displays a window that provides overview information of the trace data plotting in the CacheTune graph. The Trace Information window is shown in Figure 12. The window tells you the data cache is two-way set-associative with a total size of 0x4000 (16 K) bytes. Each cache way is 8K bytes and the cache line is 64 bytes. L1D is read-allocate cache: it only brings data into cache on a read miss only; write misses are passed directly to L2 through a write buffer, bypassing L1D. Write misses do not stall the CPU unless the write buffer is full. Unlike L1D, L2 cache is a read and write allocate cache, meaning the data is brought into cache on a write miss as well. This information is necessary to understand the cache behavior with this particular example. The cache events collected and the graph properties (cycle interval, address ranges, etc.) are also displayed in the window.

Click “OK” to close the window.

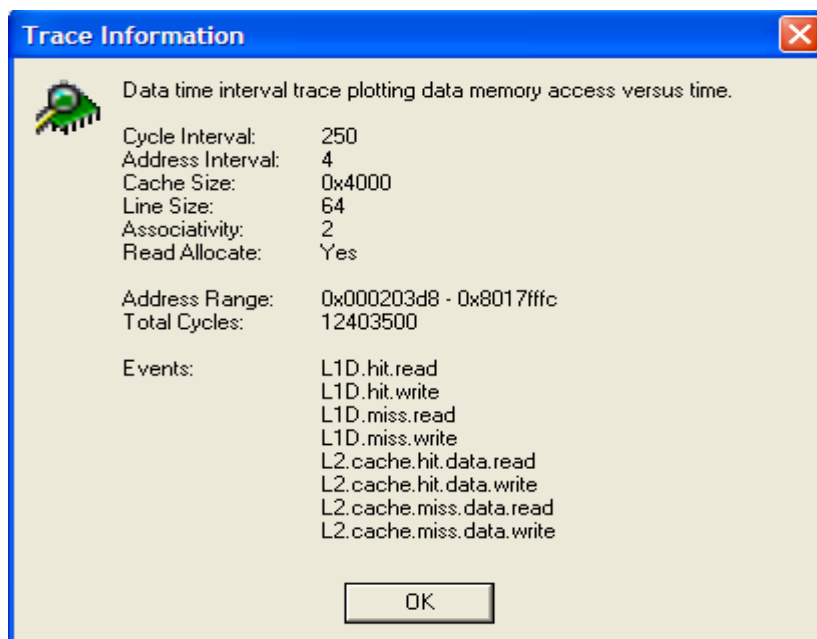



Figure 12. Trace Information Window

3. Click the Full Zoom button  to zoom out as far as possible on the address data plotted on the vertical axis and cycle data plotted on the horizontal axis. This allows you to view all data cache accesses and provides an overview of memory access patterns so that you can easily point out the hot spots where most cache misses occur. It should resemble the image shown in Figure 13. (Actual image may vary depending on screen resolution.)

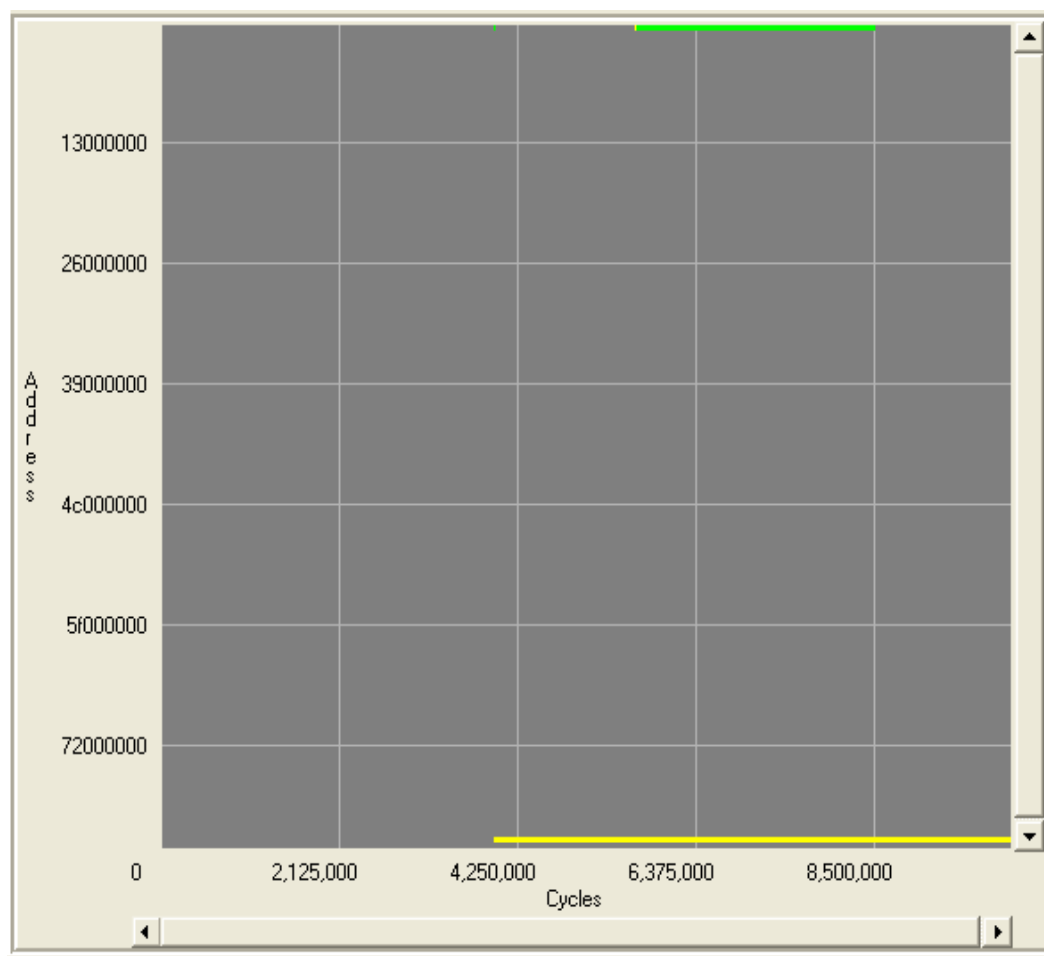



Figure 13. Overview of Data Cache Accesses

The plot of data memory accesses in the graphical display area shows up as two narrow bands, which are scattered in the upper and lower portion of the display. This happens because data are allocated in both internal and external memory and hence covers a large range. You may also observe that in the upper portion of the graph there are mostly green pixels, indicating cache hits; whereas, yellow pixels are shown in the lower portion, indicating both cache hits and misses occur. As you may recall, most data buffers that are processed by the `proc_chain` function are in the external memory; therefore, you will focus on the access to the external memory for the time being.

4. Use the zoom in area feature  to zoom in the area for the lower band. You may need to zoom in a few times to narrow down the address range in order to view in more detail.
5. From the Mark Symbol drop-down menu, select `<<All Symbols>>`. This marks all the global data symbols and their address ranges in the address section of the display. The

graph should resemble the image in Figure 14. As observed from the display, data buffers allocated in the external memory are `in_image_ext`, `expand_out`, `wave_out` and `out_image_ext`; cache events occur on memory accesses to these data buffers.

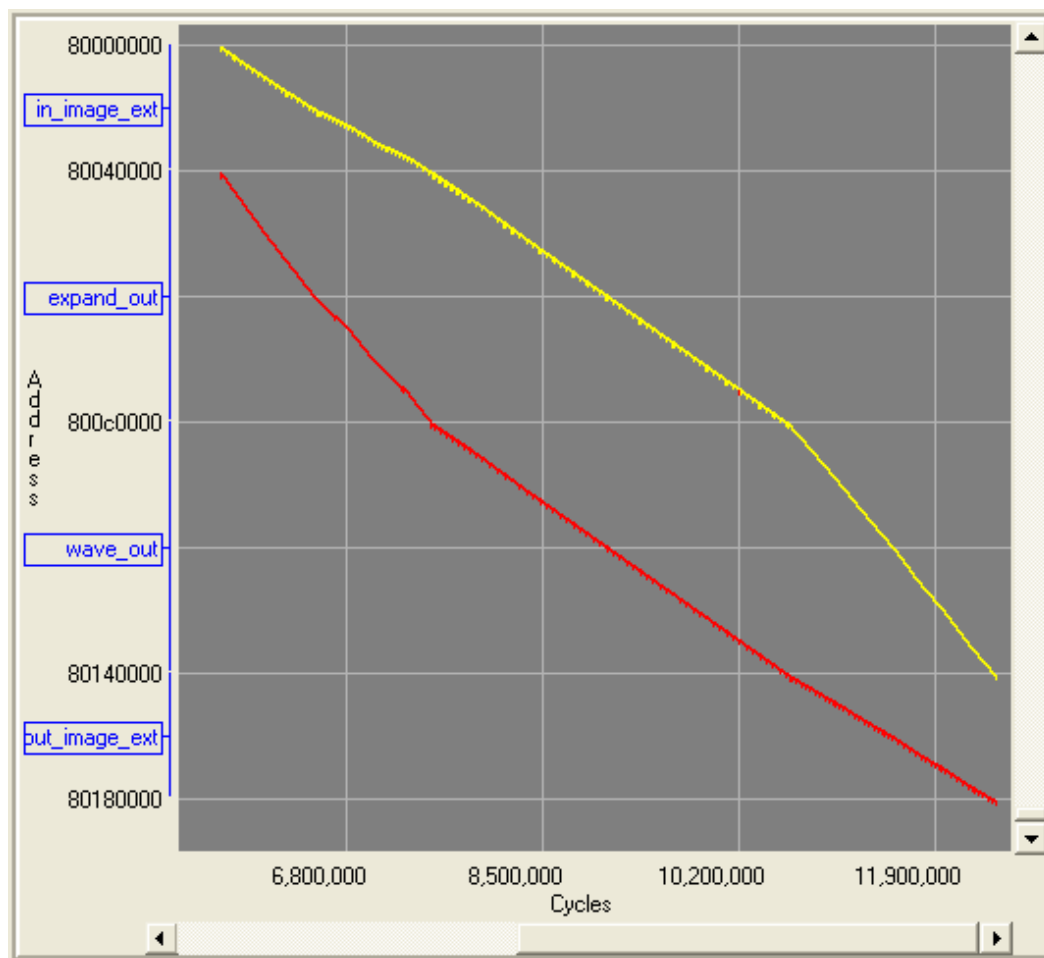


Figure 14. Sequential Accesses to Data Buffers


Also notice from the color options section on the left of the display, the default color option is on hits/misses. This color scheme color codes the memory references according to whether they were L1D cache hits or L1D cache misses. The legend area above the graph shows the color key to each cache event; for example, yellow indicates both cache hits and misses occur; red means only cache misses occur.

6. Toggle the color options to reads/write, where the references are color coded according to whether data are read from or written to memory. Notice that there are no more yellow pixels in the display but only green and red pixels. Within a particular period of time, the program processes only two buffers: reads from one buffer and writes to the other one. For example, the program initially reads data from `in_image_ext` while writing data to `expand_out`, then the program reads the data from `expand_out` and writes to `wave_out`, etc.
7. Reset the color options to miss/hits and double-click the symbol `in_image_ext` displayed in the address range area. CacheTune zooms in to display the memory range for the

in_image_ext only. This allows you to drill down the memory access patterns on selected symbol.

- From the Event Options section, deselect other cache events and leave only one event at a time. For instance, only select L1D.hit.read. This will display only the cache hit event when reading from L1D. Notice that when selecting only L1D.hit.write or L1D.miss.write, there is no event displayed in the graph. This implies that the program does not write to in_image_ext, which matches the analysis from step 6. You can also observe that read hits and read misses occur when accessing the data from L2 cache.

NOTE: The L2 events are displayed according to the order shown in the Event Option area. For example, when both L2.cache.hit.data.read and L2.cache.miss.data.read.miss occur in the same pixel, only L2.cache.hit.data.read will show up in the display. To view all L2.cache.miss.data.read.miss events, you need to deselect all the events that are in the higher order in the Event Option.

- Click the undo zoom icon  to return to previous display. Then repeat step 7 and 8 for the rest of data buffers.

For expand_out and wave_out, the entire data buffers are referenced twice. At first, the program writes to them, causing write misses in L1D and both write misses and hits in L2. And then, the program reads from them, causing read misses and hits in L1D and L2. The data accesses patterns of these two buffers are displayed in Figure 15. Again, red pixels indicate cache misses; yellow pixels indicate both cache hits and misses occur.

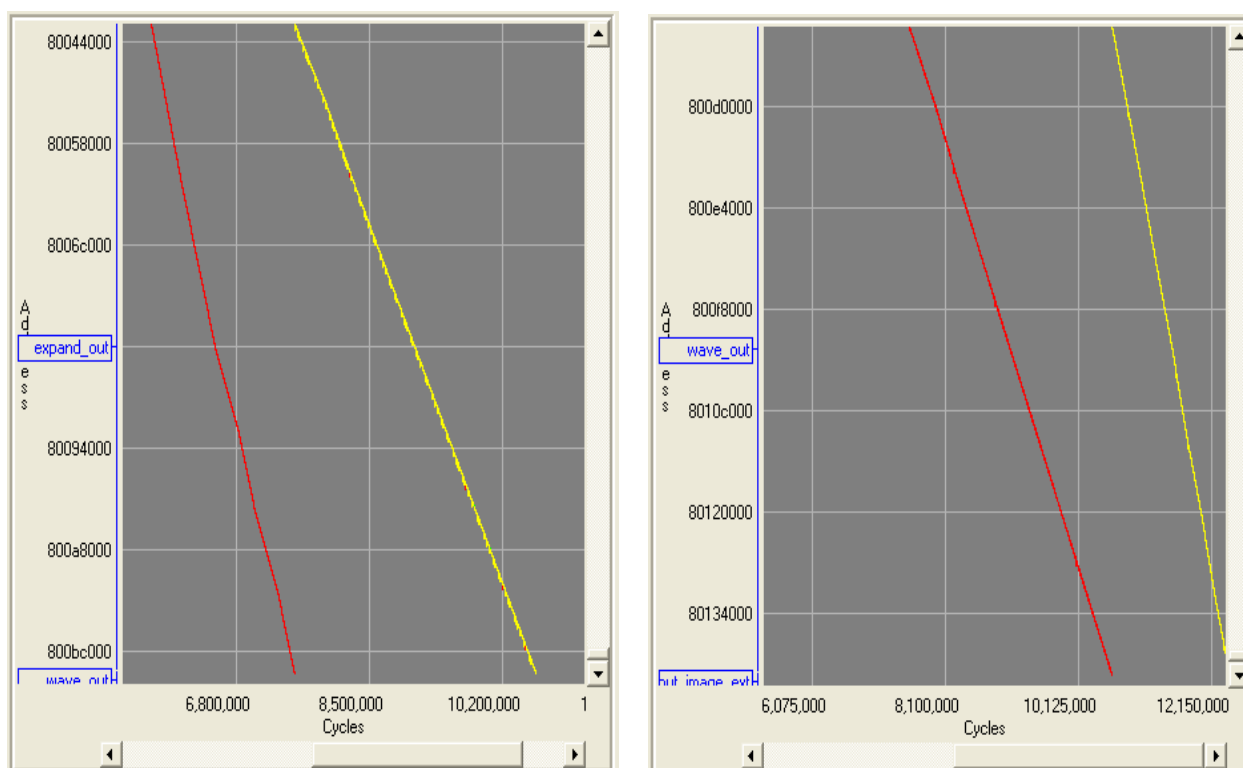


Figure 15. Access Patterns to Data Buffers

For out_image_ext, the program only writes to it, causing write misses in L1D and write misses and hits in L2.


The program makes a large series of sequential accesses to all the data buffers. This reflects a linear access pattern in the CacheTune display. When a read miss occurs, a cache line worth of data will be brought into the cache. Since array elements in each buffer are read consecutively, the subsequent accesses will hit in cache. As can be observed from the display, there are multiple consecutive read hits following one read miss in both L1D and L2 since they are both read-allocate. As L2 is also write-allocate, L2 write access has a similar pattern. L1 write miss does not allocate data in the cache, so all the writes will miss in L1D.

An L1D miss is serviced by the L2 cache. When it also misses in L2, the CPU is stalled while the L2 retrieves the data from external memory. The number of stall cycles may vary depending on type and width of external memory, as well as other aspects of system loading. The L2 miss overhead can be significant because the L2 cache needs to communicate with slow off-chip memory. Considering the speed disparity between the processor and off-chip memory, handling data transfer carefully is one of critical factors for attaining higher performance.

Step B: Visualizing and analyzing cross cache

You have known so far which data are referenced and what kind of references (hits/misses; read/write), but there is no knowledge on which functions/program accesses these data. Cross cache can help at this point.

The cross cache view in CacheTune encodes a cross-reference of program memory accesses with data cache accesses. The addresses are the same as for the program cache display, but the events displayed are with respect to the data memory reads and writes (loads and stores) contained within those instructions and whether those loads and stores hit or missed in L1D.

1. Switch to the cross cache view by clicking the Cross Cache tab located below the CacheTune toolbar. Each tab maintains its own toolbar state, mode, selected symbol, etc. The advice will change to cross cache advice.
2. Click the full zoom button . The graph will look similar to the following image. There are three horizontal bars across the display, which represent the code that references the data memory.

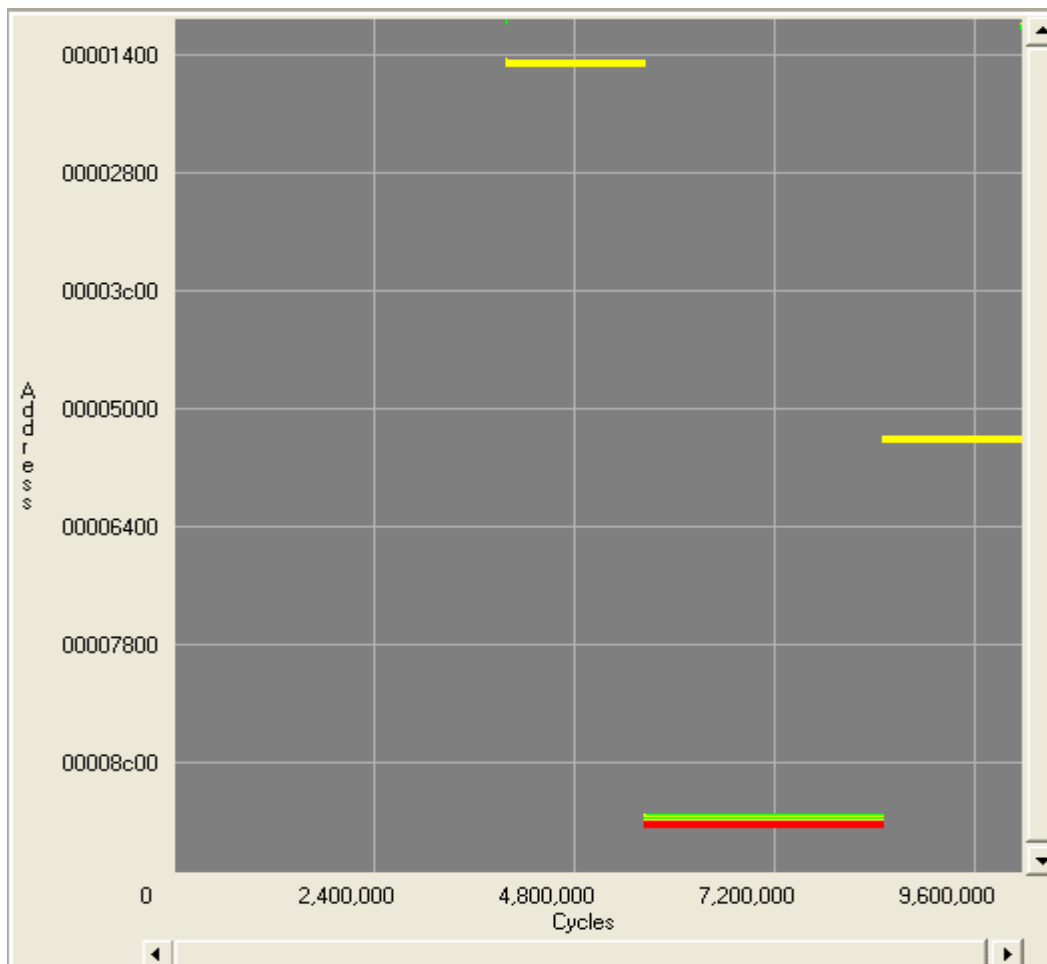


Figure 16. Cross Cache

3. Move the cursor on top of the bars. The information display area (left side of the CacheTune window) updates with the current address range, section, symbol (if any), and cycle range corresponding to the pixel currently under the cursor. This identifies three functions (from left to right): IMG_pix_expand, IMG_wave_horz and IMG_pix_sat. The yellow pixels indicate that all three functions contain executed load and store instructions. The functions are executed one at a time. You can tell what data buffers are accessed by each function by correlating execution period from x-axis across the data cache and cross cache display.


In conjunction with the analysis on data cache display, you can get the entire picture of the data flow on the processing chain. All three functions operate out-of-place, that is, the results are placed in an array different from the input. For instance, function IMG_pix_expand reads data from the input image buffer in_image_ext and then stores the results into the buffer expand_out. Also notice the results from one function forms the input of the next function. This type of processing chain scenario is often frequently found in typical DSP applications.

A good strategy for tuning cache performance is to start with the application level then move to the procedural level. The following steps will reflect the order to address the optimization.

3.3.1 Applying Application Level Optimization: EDMA Double Buffering Framework

In this section, you will apply the cache optimization techniques at the application level to reduce the data cache related overhead. The application level cache optimization tends to be straightforward to implement and typically have a high impact on the overall performance improvement.

Step A: Examining the general optimization advice

1. Use the back arrow on top of the advice window to go back to the CacheTune General Advice. Click the yellow arrow  next to Optimization techniques. This opens up the related topic in online help which discusses in detail on the general optimization techniques.
2. Go through the text under this topic, noting that it is recommended to apply application level optimizations first then procedural level. Also note one application level technique is to use EDMA for transferring from external memory instead of L2 Cache.

For this application, data is too large to fit in L2 SRAM and has to be allocated into external memory. Using L2 as cache is a fast way to get an application up and running as the off-chip memory access is seamlessly handled by the cache controller with L2 cache. You can also use L2 SRAM with EDMA to transfer data to external memory. Whereas the EDMA transfer involves more programming effort, it is typically more advantageous than using L2 cache alone in terms of performance.

Step B: Using EDMA double buffering framework

Figure 17 shows the program flow for an EDMA double-buffering framework for the wavelet processing chain. The image is transferred to a buffer in L2 SRAM using EDMA. The processing chain is then processed, and the output is transferred back to the external memory. While one EDMA input buffer is being processed, EDMA is filling the second buffer in the background.

When implementing the EDMA double buffering framework, one important factor to consider is the size of the EDMA buffer. The buffer should be large enough to minimize EDMA setup cycles. Or, it should be small enough to leave enough on-chip memory for other critical data and code. In this example, a buffer size of 16K bytes is chosen. As smaller buffers are processed at a time, the interface buffers (expand_out and wave_out) can be allocated into L2 SRAM. The input image is 256 Kbytes, so 16 EDMA transfers are required.

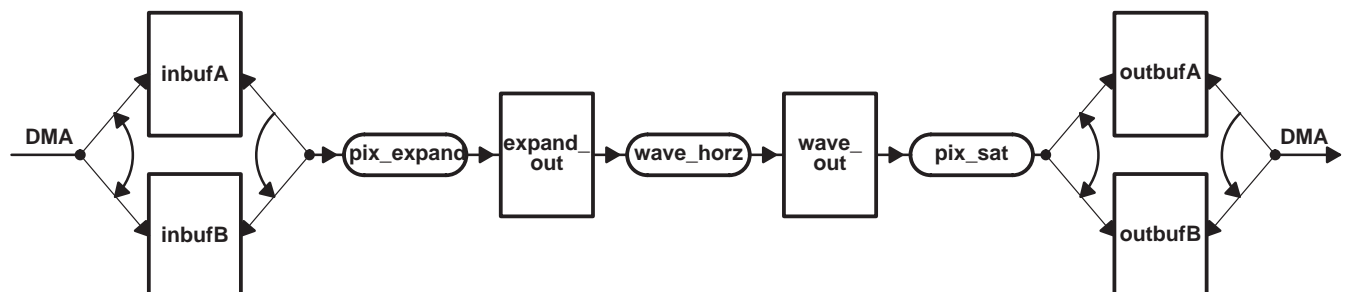


Figure 17. EDMA Double Buffering Framework

1. From Windows Explorer, copy the files from the stage1 folder to your project folder: c:\ccstudio\wavelet\original.

2. Click Yes to replace the existing files. The new files include a new function called `dma_double_buf` that implements the EDMA double buffering framework to transfer the data from external memory instead of L2 cache. The routines used for setting up the EDMA transfers (for example, `DAT_copy` and `DAT_wait`) are from the C64x Chip Support Library (CSL). Refer to *TMS320C6000 Chip Support Library API Reference Guide* (SPRU401) for more information.
3. Rebuild the project.
4. Perform a CPU Reset, and reload the program.

Step C: Resetting the range of profile data

As the program is modified, more functions are added and the profiled data are changed. You need to reset the range of profile data using the Profile Setup tool.


1. Re-open Profile Setup from Profile menu.
2. Click the Range tab and observe that several new functions have been added into the function list. The function list automatically re-populates to include the latest functions after loading the program.
3. The positions for the control points set in the source file `main.c` are no longer applicable as the program is changed. Select the halt collection point from the Control tab and hit “delete” key to remove it. Follow the same procedures to remove the resume collection point.
4. Repeat the steps described in section 3.2 step B to set halt collection points at the source line that calls `DAT_close` in the file `main.c`; a resume collection point at the source line that calls `dma_double_buf` function.

NOTE: The halt collect point set at the symbol `c_int00` can be left unchanged as it is created by symbol name and the position is updated automatically.

5. Hide the Profile Setup window.

Step D: Measure the cache usage improvement

1. Run the application to completion.
2. To verify that correct output is generated, you can check the Graphical Display in Standard Layout. Click the Standard Layout icon to switch back the standard layout.
3. Right-click the Graphical Display window of the output image and select refresh from the context menu.
4. Once you have verified that the same image as shown in Figure 8, click the Tuning wrench icon to activate the tuning layout again.
5. Inspect the profile data in the inclusive columns of the `dma_double_buf` function as now most subroutines are called by this function.
6. From Profile Viewer, save this profile data set as `run1.xml` in dataset folder that is created in section 3.2 step C.
For comparison, you can open another Profile Viewer window to view the previously saved data set.
7. From Profile menu, select Viewer to launch another Profile Viewer.

8. From the Profile Viewer toolbar, click the load data set button  and open previously saved data set run0.xml. This displays the saved data set in the new Profile Viewer window.

The comparisons of profile results before and after optimization are summarized in Table 3. Close the Profile Viewer of run0.xml after comparing the data.



Table 3. Profile Data Comparison 1

| Profile Events | Stage 0 | Stage 1 |
|--------------------------|-----------|---------|
| CPU.stall.mem.L1D | 6,068,428 | 222,072 |
| CPU.stall.mem.L1P | 132 | 1,487 |
| Cycle.CPU | 640,262 | 651,038 |
| Cycle.Total | 6,709,606 | 874,298 |
| L1D.stall.write_buf_full | 2,722,684 | 5,653 |
| Cache Overhead | 947% | 25.6% |

9. Check the Delta column of Goals Window. It shows a decrease of 5 million (5,835,303) cycles. This is almost an 8x improvement over the previous implementation by applying the application level optimization.

The application benefits from processing the data buffers that are allocated in L2 SRAM. This reduces cache overhead and gives you more control over memory accesses since only Level 1 cache is involved whose behavior is easier to analyze. Meanwhile, this allows you to make some modifications to algorithms in the way the CPU is accessing data, and/or to alter data structures to allow for more cache-friendly memory access patterns. Your numbers may vary.

3.3.2 Applying Procedural Level Optimization: Restructuring the Data Layout

1. From CacheTune toolbar, click the Data Cache tab to display the updated data cache accesses.
2. Click the Full Zoom button  to view all the data accesses. Observed most data cache accesses occur in L2 SRAM memory area.
There are two narrow band of cache misses occur at the memory range 0x01A00000 – 0x02000000. These memory ranges are for the EDMA control registers. Memory-mapped peripheral registers are non-cacheable and therefore accesses to them will miss in cache. The data will pass directly to the CPU and are not stored in cache. These misses are considered as compulsory misses.
3. Use the Zoom in Area button  to select the area where the data buffers are referenced.
4. From the Mark Symbol drop down menu, select <<All Symbols>> to display all the symbols. The image should resemble Figure 18.

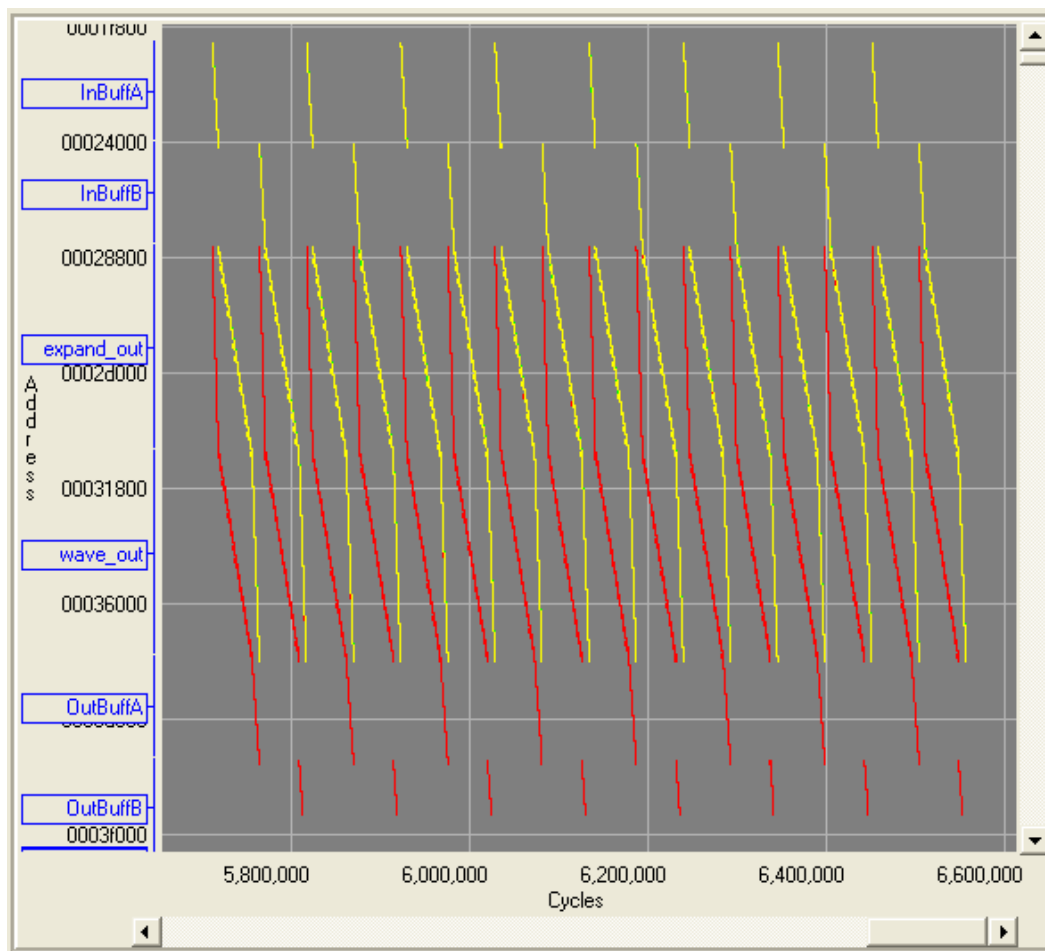



Figure 18. Data Cache Accesses After Using EDMA

As may be observed from Figure 18, similar linear processing chain access patterns were repeated 16 times (16 EDMA transfers). Read misses and write misses occur when accessing the data, causing L1D stalls as well as write-buffer-full stalls. Data buffers `expand_out` and `wave_out` are both 32K bytes in size, and are larger than the cache size 16K bytes. After each iteration, the beginning of the data buffer has already been replaced with the end of the data buffer as the capacity is insufficient. The following iteration will then experience misses again.

5. Examine the Data Cache Advice, noticing that this matches the third miss scenario. Click the link  next to the typical miss patterns. This brings up the online help window, displaying a typical CacheTune graph for this kind of miss patterns. Also notice the figure looks very similar to your CacheTune graph.


Based on the analysis above, most read misses on L1D are capacity misses due to sub-optimal data usage, that is, the algorithm does not reuse the data when they are still in cache. The Data Cache Advice prompts you that this type of miss can be reduced by restructuring the data in order to work on smaller blocks of data at a time.

In a processing chain, the output of one function forms the input of the next function. If the buffers that interface the functions (expand_out and wave_out) can be retained in L1D, there will be no cache misses inside the processing chain, thus completely eliminating read miss and write buffer related stalls. An appropriate memory layout has to be carefully arranged to ensure the interface buffers are retained in L1D. In this example, the program has read accesses to four buffers: the two input EDMA buffers and two interface buffers (expand_out and wave_out). To fit the interface buffers completely in cache and protect them from being evicted by the input buffers, they need to be placed within one cache way (8K bytes). This leaves the interface buffer with a size of 4K bytes each and they have to be allocated contiguously in memory.

6. From Windows Explorer, copy the files from the stage2 folder to your project folder: c:\ccstudio\wavelet\original. Click Yes to replace the existing files.
7. You can open the new proc_chain.c source file to view the data declaration to fulfill the memory layout requirement. Rebuild the program.
8. Reset CPU then reload the program.
9. Run the program to the completion.
10. Repeat the steps discussed earlier to validate the application by viewing the image output.
11. Check the delta column of Goals Window. Notice there is a decrease of 149,110 cycles, a 17% improvement from the previous run.
12. The Profile Viewer window also updates to display the profile data. Check the results of the dma_double_buf function. Save the profile data set as run2.xml. You can repeat the steps discussed in section 3.3.1 Step D to open another Profile Viewer window to compare the numbers. Table 4 gives a summary on comparisons between run2 and run3. Note that the L1D stalls including write buffer stalls are reduced dramatically, but L1P stalls increases. The cache overhead is reduced to 11.1%. Close the Profile Viewer of run2 after comparing the data.

Table 4. Profile Data Comparison 2

| Profile Events | Stage 1 | Stage 2 |
|--------------------------|---------|---------|
| CPU.stall.mem.L1D | 222,072 | 57,922 |
| CPU.stall.mem.L1P | 1,487 | 10,352 |
| Cycle.CPU | 651,038 | 657,598 |
| Cycle.Total | 874,298 | 725,198 |
| L1D.stall.write_buf_full | 5,653 | 42 |
| Cache Overhead | 25.6% | 10.4% |

13. Click the data cache tab of CacheTune and then click on the Zoom in Area button  to view the data cache accesses again. A screen shot is shown in Figure 19.

The function now works on smaller pieces of the input buffer at a time. This reduces the size of the working set and improves temporal locality. The only read misses that then occur are compulsory misses for the first routine, IMG_pix_expand, which reads new data from the EDMA buffers. The first time the processing chain is executed, read accesses to these interface buffers will miss. However, all following iterations will then access these buffers in L1D.

Note that repeated write misses on the EDMA output buffers (OutBuffA and OutBuffB) in L1D that does not write-allocate are categorized as compulsory misses.

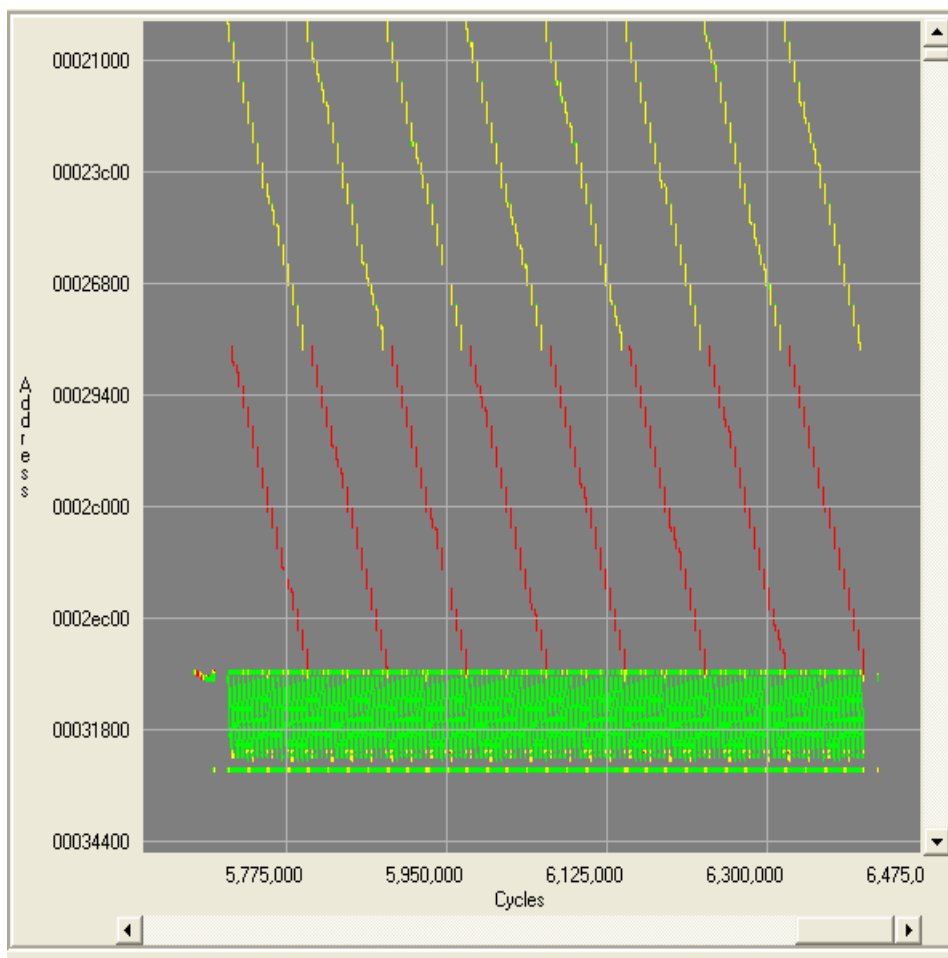


Figure 19. Data Cache Accesses After Restructuring

3.3.3 Exploiting Miss Pipelining

The C64x cache architecture pipelines read misses, which reduces the read miss overhead by overlapping processing of several cache misses. Multiple parallel and consecutive read misses consume as few as 2 cycles once pipelining is setup:

1. Examine the Data Cache Advice. The last advice is related to touch loop. The touch routine reduces the number of stall cycles per miss by exploiting miss pipelining. The previous discussion showed how to eliminate most of the cache misses. The only cache read misses left are now the compulsory misses due to the first time reference of the input data. The impact of compulsory misses can be reduced by using the touch routine. Also, the touch routine can be used to pre-allocate the interface buffers prior to the first iteration. This has the additional effect of eliminating write buffer related stalls during the first iteration.


2. From the Data Cache Advice window, click the link  under the touch loop advice. The online help provides the complete source code and C prototype for the touch routine. For your convenience, the source code of touch routine is also provided in folder stage3 as file touch.asm.
3. Click the link next to the example for the third miss scenario. The example in the online help also has a reference on how to utilize the touch routine.
4. Copy the files in folder Stage3 into your project directory. The file touch.asm contains a C callable assembly touch routine.
5. Add the file touch.asm into the project and rebuild the project.
6. Reset the CPU and reload the program.
7. Run the program to completion.
8. Validate the application by inspecting the output image.
You can now visualize the new data cache access pattern with the CacheTune tool and use the profile tools to check the improvement.
9. From CacheTune graph, notice the series of read misses occur when reading from input buffers and the first accesses to the interface buffers. These consecutive misses are pipelined, and thus the overall miss stalls are reduced.
10. From Profile Viewer, save the data set as run3.xml. Repeat the steps described in last section to open another profile viewer and compare the profile data from the previous run. Table 5 gives a summary on the cache events for function dma_double_buf. Notice that the L1D stalls have decrease 21, 914 cycles and the overall application cycles reduces 17,052 cycles. Your numbers may vary.


Table 5. Profile Data Comparison 3

| Profile Events | Stage 2 | Stage 3 |
|--------------------------|---------|---------|
| CPU.stall.mem.L1D | 57,922 | 36,008 |
| CPU.stall.mem.L1P | 10,135 | 10,484 |
| Cycle.CPU | 657,598 | 662,082 |
| Cycle.Total | 725,198 | 708,146 |
| L1D.stall.write_buf_full | 42 | 0 |
| Cache Overhead | 10.4% | 7% |


3.4 Program Cache Visualization and Optimization


While the data cache stalls are dramatically reduced, more program cache stalls occur. You need to investigate the program memory access pattern and improve program cache utilization.

Step A: Visualizing and analyzing the program cache

1. Click the Program Cache tab.
2. Click the Full Zoom button  to view the entire trace.
3. Hover your mouse pointer over the cache misses legend. This view only the cache misses event, while the other events are grayed out.

Notice there are three red horizontal bars across the display. This signifies that the same piece of code repeatedly misses the cache within a short period of time.

4. Examine the program cache advice. There are two possible miss scenarios: conflict misses and capacity misses. Click the arrow  next to the Typical Patterns under conflict misses. This opens up the online topic which displays the typical miss patterns for conflict misses. The miss patterns are very similar to the one shown in the CacheTune display. This signals that the cache misses might be caused by memory conflict.
5. The interference shadow feature supplied with the CacheTune tool can easily highlight this kind of miss. Select the origin of the grid at an address within the memory range of function IMG_pix_sat (represented by the horizontal bar in the middle). The resulting display will show dark bands across the graphical display area to indicate the address ranges that interfere with this function in the cache.

When you select the Interference Shadow, it will display the symbol name in the Address Section; this also enables the Find Conflicting Symbols button . Find Conflicting Symbols displays a list of the symbols and graphs that conflict in memory with the selected symbol.

6. As the symbol name is too long to be displayed in the address area, only the last 12 characters are shown. To view the entire symbol, simply roll your mouse point on top of the symbol box and the complete symbol name will be display in the tip box, as shown in Figure 20.

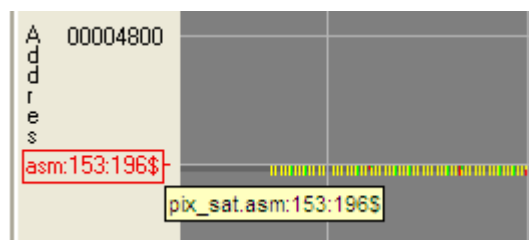



Figure 20. Viewing the Symbol

7. Click the Find Conflicting Symbols button . This displays all cache conflicting symbols with a red box around them in the Address Section. Also, the CacheTune tab of Advice Window displays text information on the conflicting symbols, such as the symbol's name, the conflict range, etc. Again, hover your mouse over the cache misses legend to view only the cache misses event. A screen shot is shown in Figure 21. Notice dark shadows are drawn on top of the upper and lower horizontal bars.

These repetitive misses are due to cache conflict misses, which result from three functions, IMG_pix_expand, IMG_pix_sat, and IMG_wave_horz, being mapped to the same cache line of L1P (L1P on C6416 is direct-mapped). When the functions execute back to back (they are called repeatedly in a loop), they replace each other from the cache and miss the next time they execute.

NOTE: In order to view the symbols of functions from Image Library, you might need to rebuild the library. To rebuild the library, using the following command from the LIB directory where image library is installed:

```
mk6x img64x.src -mv6400 -l img64x.lib
```

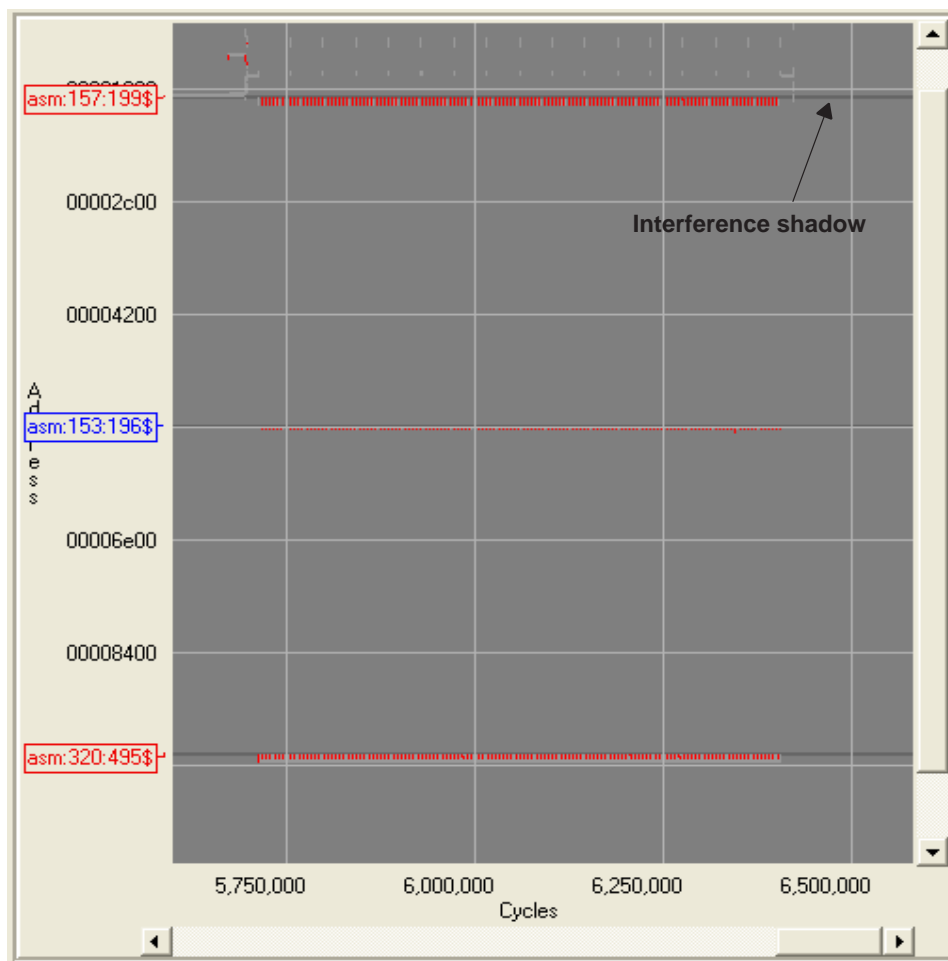


Figure 21. Conflict Misses in Program Cache

Step B: Eliminating the conflict misses

The cache misses are called conflict misses because two or more functions are mapped to the same cache lines. The affected functions can be placed in different memory locations to ensure they do not overlap in the cache. CacheTune provides general advice and example on how to eliminate these kind of misses.

1. Click the Program Cache tab again to bring back the program cache advice. Click the yellow arrow next to Example under conflict misses category. This brings up the online topic which uses an example to describe possible solutions to eliminate the conflict misses. One possible solution is to allocate the functions contiguously in memory. Since the functions that execute when the interference occurs occupy less than 16KB of memory (the C64x program cache size), allocating them contiguously will eliminate any conflicts. In addition to the three functions in the processing chain, the code for setting up EDMA transfers (for example, DAT_copy and DAT_wait routines) and the touch loop should also be taken into account. This can be accomplished by editing the linker command file. For details regarding the linker command file, refer to *TMS320C6000 Assembly Language Tools User's Guide* (SPRU186).

2. Replace the linker command file with the one in folder stage4. A portion of the linker command file is shown in Figure 22. The functions are not only allocated contiguously, but also allocated in the way sorted by the CPU cycle counts. This is to guarantee that the functions that execute the most cycles are allocated together so as to avoid potential conflict misses. A GROUP command guarantee the functions are allocated in the order specified in the linker command file.

```

SECTIONS
{
    GROUP          >          SRAMT
    {
        .text:_wave_horz
        .text:_pix_expand
        .text:_pix_sat
        .text:_touch
        .text:_DAT_copy
        .text:_DAT_wait
        .text:
    } /* End of GROUP */
    . . .
}

```

Figure 22. Modified Linker Command File


3. Re-link the application with the new function placement.
4. Reset CPU and reload the program.

Step C: Measure the cache usage improvement

1. Run the program to completion. Once program finishes execution, the CacheTune output window, Goals Window, and Profile Viewer will update to display the new data.
2. View the new profile results of function dma_double_buf from Profile Viewer window.
Table 6 gives a comparison of profile data before and after the optimization. The stalls caused by L1P misses are remedied. This results in an improvement of the application execution time by 9,474 cycles. Although in this case this is not as significant as the improvement from the data cache optimization, this is simply achieved by re-linking the program and without modifying the code and recompilation.
3. Save the profile data set as run4.xml from Profile Viewer.

Table 6. Profile Data Comparison 4

| Profile Events | Stage 3 | Stage 4 |
|--------------------------|---------|---------|
| CPU.stall.mem.L1D | 36,008 | 36,355 |
| CPU.stall.mem.L1P | 10,484 | 269 |
| Cycle.CPU | 662,082 | 662,070 |
| Cycle.Total | 708,146 | 698,672 |
| L1D.stall.write_buf_full | 0 | 14 |
| Cache Overhead | 7% | 5.5% |

4. Within CacheTune, select the program cache tab; then click the Full Zoom button . Figure 23 gives the display of program cache after the optimization.

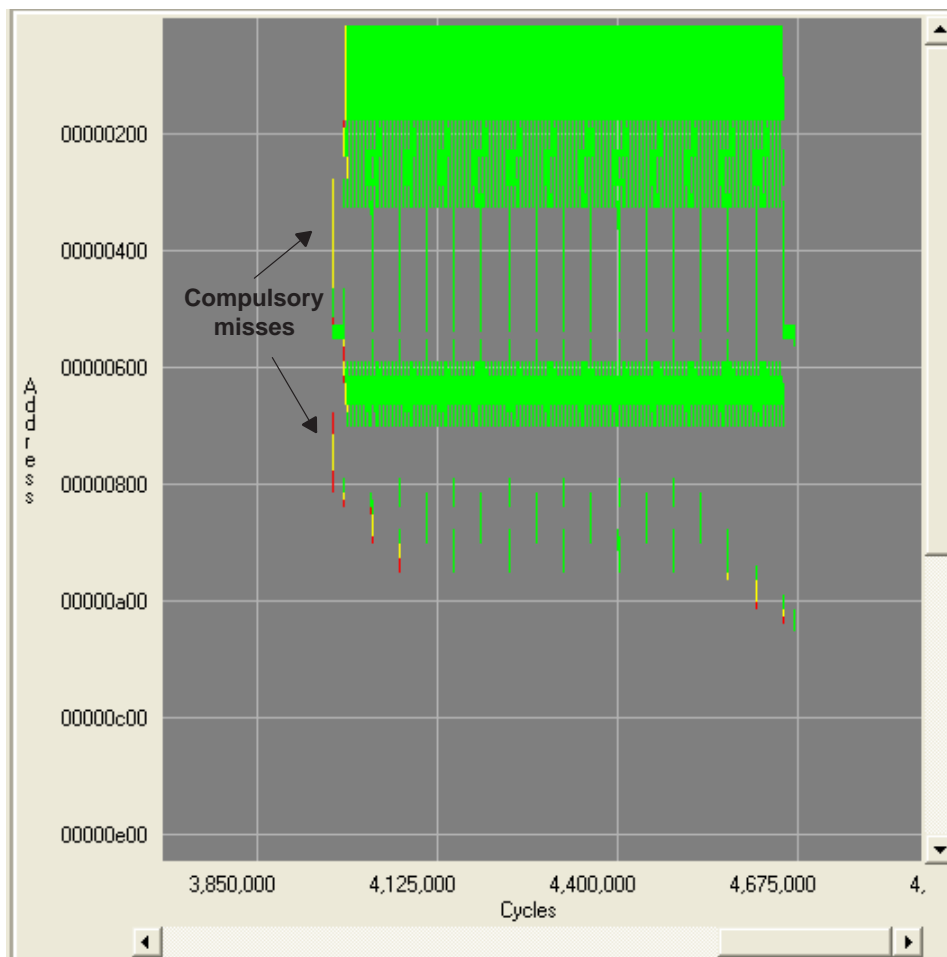


Figure 23. Program Cache After Optimization

From the display, notice all functions are now grouped together and the conflict misses are eliminated. The cache misses that are left are compulsory misses.

3.5 Overall System Improvement

All the profile data through Table 3 to Table 6 are consolidated into Table 7 for a breakdown of results after each stage. Another way to do this is to export each saved data set into delimited text file from Profile Viewer; then import them to a spreadsheet program.

By optimizing the cache effectiveness, almost a 10x reduction in application total cycle counts is obtained and the cache overhead is reduced from 947% to 5.5%! Most improvement is from eliminating data cache misses and hence the related miss stalls and write buffer full stalls. The application level optimization of using DMA double buffer framework has the biggest impact on the performance. The data cache stalls are further reduced significantly by applying procedural level optimization techniques. The capacity misses in data cache are lessened by re-structuring the data to work on smaller buffers at a certain time and the stall cycles for compulsory misses are minimized by exploiting miss pipelining. The data cache tuning requires the function to be executed repeatedly. As functions are overlapped in the program cache, this cause conflict misses in the program cache and hence an increase of the L1P stalls. The conflict misses are completely eliminated by allocating the program contiguously in memory.

Table 7. Overall Cache Events Comparison

| Profile Events | Stage 0 | Stage 1 | Stage 2 | Stage 3 | Stage 4 |
|-------------------|-----------|---------|---------|---------|---------|
| CPU.stall.mem.L1D | 6,068,428 | 222,072 | 57,922 | 36,008 | 36,355 |
| CPU.stall.mem.L1P | 132 | 1,487 | 10,135 | 10,484 | 269 |
| Cycle.CPU | 640,262 | 651,038 | 657,598 | 662,082 | 662,070 |
| Cycle.Total | 6,709,606 | 874,298 | 725,198 | 708,146 | 698,672 |
| Cache Overhead | 947% | 25.6% | 10.4% | 7% | 5.5% |

4 Conclusion

Texas Instruments' new CacheTune tool offers a plethora of useful features and proactive advices to suit the tuning need of maximizing cache effectiveness.

The example walked through the steps necessary to utilize the CacheTune tool. The tool helps you quickly identify performance overhead in the data cache. The overhead in the data cache are reduced drastically after applying both application level and procedural level optimization. CacheTune is also very effective in detecting functions conflicting with each other in the program cache. You easily remedy these conflict misses by reallocating the conflicting functions using the linker command file. The misses in the program cache are eliminated completely without any extra coding effort and compilation. The overall system performance is boosted up by almost 10x after using the tool to tune the cache effectiveness.

The CacheTune tool addresses the aspects of efficiency analysis targeted at better memory management and helps developers achieve efficiency quickly, hence shortening the application development lifecycle and reducing the time to market for the developer.

5 References

1. *TMS320C64x DSP Two-Level Internal Memory Reference Guide* (SPRU610)
2. *TMS320C621x/C671x Two Level Internal Memory Reference Guide* (SPRU609)
3. *TMS320C6000 DSP Cache User's Guide* (SPRU656)
4. *TMS320C64x Image/Video Processing Library Programmer's Reference* (SPRU023)
5. *TMS320C6000 Chip Support Library API Reference Guide* (SPRU401)
6. *TMS320C6000 Assembly Language Tools User's Guide* (SPRU186)
7. *Cache Usage in High-Performance DSP Applications With the TMS320C64x* (SPRA756)
8. David A. Patterson and John L. Hennessy, *Computer Organization & Design*, Second edition, Morgan Kaufmann, 1998

Appendix A Cache Basics

Processing data at high clock rates requires fast memory connected directly to the CPU (Central Processing Unit). However, a bandwidth dilemma has occurred with the dramatic increase in processor speed. While processor speed has increased dramatically, memory speed has not. Therefore, the memory to which the CPU is connected often becomes a processing bottleneck. Caches are small, fast memory that reside between the CPU and slower system memory. The cache provides code and data to the CPU at the speed of processor while automatically managing the data movement from the slower main memory that is frequently located off-chip.

This section will introduce the basic conceptual ideas behind cache, though many abstractions are used for simplification. Refer to *TMS320C6000 DSP Cache User's Guide* (SPRU656) for a more detailed discussion.

Cache operates by taking advantage of the principle of locality. There are two different types of locality:

- Temporal locality: if an item has been accessed recently, it is likely to be accessed again. Accessing the instructions and data repeatedly within a loop structure shows a high amount of temporal locality.
- Spatial locality: items that are close to other recently accessed items are likely to be accessed soon. For example, sequentially accesses to matrix elements will have high degrees of spatial locality.

Cache memory takes advantage of locality by holding current data or program accesses closer to the processor. The smallest block of data that the cache operates on is called a line. Typically, the line size is larger than one data value or one instruction word in length.

If data from a requested memory location appears in a line of cache, this is called a hit. The opposite event, a miss, occurs when the requested data is not found in the cache. If a miss occurs, the next level of memory is accessed to fetch the missing data. The number of cache misses is often an important measure of cache performance; the more misses you have, the lower your performance will be. In addition when data is missed, a location needs to be selected to place the newly cached data. This process is known as allocation, which often involves replacing the data occupying an existing cache line to make room for the new data.

Cache can be categorized by the schemes used for placing lines. A direct-mapped cache maps each line of memory to exactly one location in the cache. This is in contrast to a multi-way set-associative cache, which selects a "set" of locations to place the line. The number of locations in each set is referred as the number of ways. For instance, in a 2-way set-associative cache, each set consists of 2 line-frames (ways). Any given cacheable address in the memory map maps to a unique set in the cache, and a line can be placed in two possible locations of that set. An extreme of set-associative cache is fully associative cache that allows any memory address to be stored at any location within the cache. For the latter two types of cache, an allocation policy will be needed to choose among line frames in a set when a cache miss occurs.

Let us now investigate the sources of cache misses and how the misses can be remedied from the programmer's perspective. All cache misses can be divided into one of these three classes:

- Compulsory misses: these cache misses occur during the first access to a line. This miss occurs because there was no prior opportunity for the data to be allocated in the cache. These are sometimes referred to as "first-reference misses".

- Capacity misses: these cache misses occur when the cache does not have sufficient room to hold all the data during the execution of a program.
- Conflict misses: these cache misses occur because more than one data or program code are competing for the same cache line.

These sources of misses can be reduced by a number of code optimization techniques. Conflict misses can be eliminated by changing the locations of data or program code in memory, and hence they will not contend for the same cache line. Capacity misses can be reduced by working on smaller amounts of data or code at a time, which can be achieved by reordering the accesses of the data or by partitioning the algorithm into smaller pieces. Refer to *TMS320C6000 DSP Cache User's Guide* (SPRU656) for more discussions on cache optimization techniques.

Appendix B Cache Structure on TI C6000 DSPs

There are several TI C6x DSP devices that contain one or more caches. An instruction cache is available on certain TMS320C620x and TMS320C6701 DSPs to buffer most recently accessed instructions on chip. Some members of TMS320C6000 DSP family employ a two-level cache for internal data and program storage to deliver high performance without the cost associated with large on-chip memory.

B.1 TMS320C6000 Two-Level Cache

Some newer members of TMS320C6000 family (C6x1x devices) employ a highly efficient two-level memory architecture for on-chip program and data accesses. In this hierarchy, the C6x1x CPU interfaces directly to a dedicated level-one program (L1P) and data (L1D) cache. These L1 caches operate at the same speed as the CPU.

The L1P operates as a direct-mapped cache. It is readable only. The L1D is a two-way set associative cache. The L1 memories are connected to a second-level memory of on-chip memory called L2. L2 is a unified memory block that contains both program and data. The L2 cache serves as a bridge between the L1 and off-chip memory. Refer to *TMS320C64x DSP Two-Level Internal Memory Reference Guide* (SPRU610) for a detailed documentation of this cache architecture.

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