Application Report
SPRA982 − December 2003

Using the Cache Analysis Tool to Improve I-Cache Utilization on C55x Targets

Ning Kang *Software Development Systems*

ABSTRACT

With the complexities of digital signal processor (DSP) applications and the system memory they utilize, it is often very difficult for the developer to understand where and when processor memory accesses are occurring. This is further complicated by the fact that many processors have incorporated cache into the on-chip memory. While there are many issues with system memory optimization, the first of those issues needing to be addressed are cache analysis, visualization, and optimization.

Maximizing cache effectiveness becomes a key to boosting the overall system performance. Cache efficiency improves processor throughput by reducing central processing unit (CPU) stall cycles due to memory activities. Cache analysis is a new tool that helps the developer achieve high levels of cache efficiency by graphically visualizing the memory reference patterns of a program over time.

This application report covers cache fundamentals, an overview of instruction cache (I-Cache) on existing TI TMS320VC55x™ DSPs, and how to make use of the cache analysis tool to achieve high levels of cache efficiency.

Note: *This document contains source code that can be accessed through this link.* *http://www−s.ti.com/sc/psheets/spra982/spra982.zip*

Contents

List of Figures

List of Tables

1 Introduction

The performance of many DSP applications is dictated by its ability to quickly access memory. With the growing complexities of system, utilizing memory effectively becomes a challenge for DSP developers to meet their real-time requirements. This is further complicated by the fact that many processors have incorporated cache into the on-chip memory. While there are many issues with system memory optimization, the first those issues that need to be addressed are cache analysis, visualization, and optimization.

The cache analysis tool, available in the analysis tool kit (ATK) in Code Composer Studio™ v2.2, aids the developer in quickly improving cache usage and thereby optimizing the CPU cycles consumed in the cache subsystem.

Cache analysis provides a way to view profiled cache events by address range over time, at different resolutions of time. It effectively helps the developer to recognize non-optimal cache usage by identifying cache misses. Issues such as functions conflicting with one another can be easily discovered. Using this tool, significant improvement in cache utilization can be achieved. It is currently targeted for the C6x1x and C55x devices.

This application report discusses how to utilize the cache analysis tool to improve cache efficiency. Section 2 covers cache fundamentals. Section 3 gives an overview of instruction cache (I-Cache) on existing TI TMS320VC55x DSPs. Section 4 introduces the cache analysis tool and describes the recommended code development flow to improve cache efficiency. How the tool fits into the application software development life cycle is also discussed briefly in this section. In section 5 an example is provided to illustrate the necessary steps required to utilize the cache analysis tool.

Advanced users that are familiar with cache concepts and cache architecture on TI C55x devices can skip sections 2 and 3.

2 Cache Basics

Processing data at high clock rates requires fast memory connected directly to the CPU. 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 the *Cache Analysis User's Guide* (SPRU575) 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's 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 the *Cache Analysis User's Guide* (SPRU575) for more discussions on cache optimization techniques.

3 I-Cache on TI TMS320VC55x DSPs

TI's TMS320VC55x (C55x) DSPs utilizes an instruction cache (I-Cache) to reduce the average access time to instruction memory and prevent repetitive external accesses by buffering the most recent instructions accessed by the CPU. Table 1 indicates the I-cache type on each C55x DSP and the available configurations for each type.

Table 1. I-Cache on C55x DSPs

C55x DSP	Available Configurations
C5501/5502	2-way set-associative cache
C5509	–
C5510	2-way set-associative cache and no Ram sets 2-way set-associative cache and one Ram set 2-way set-associative cache and two Ram sets

Ramset that exists on C5510 operates differently from two-way set associative and direct-mapped cache mode: an entire Ramset block is mapped as a single set with multiple lines. Ramset mode is more efficient for linear code that does not branch out of the set range of Ramset often. Refer to the *Cache Analysis User's Guide* (SPRU575) for detail descriptions of the operations of each of these cache types.

4 An Introduction to Cache Analysis Tool

TI's new cache analysis tool is aimed at identifying cache efficiency problem areas and visualizing them to facilitate making rapid improvements in cache performance. The cache analysis tool is based over the simulation platform. The C55x cache simulator is capable of generating cache trace file on C55x targets, which can be visualized using the cache analysis tool.

The trace file can be viewed inside the two-dimensional address-over-time display grid. All the accesses are color-coded by type (miss = red, hit = green) and various filters, panning, and zoom features facilitate quick drill down. This visual temporal view of cache accesses enables quick identification of problem areas, and whether they are conflict, capacity, or compulsory related. All of these features combine to greatly ease efforts of improving the cache efficiency of the overall application.

After analysis toolkit (ATK) is installed, the cache analysis tool can be invoked by selecting the Tools → Analysis Toolkit → Cache Analysis menu item in Code Composer Studio. Section 5 discusses the procedures to setup and invoke the tool in detail with a code example.

Figure 1 demonstrates a sample cache analysis window and some annotations.

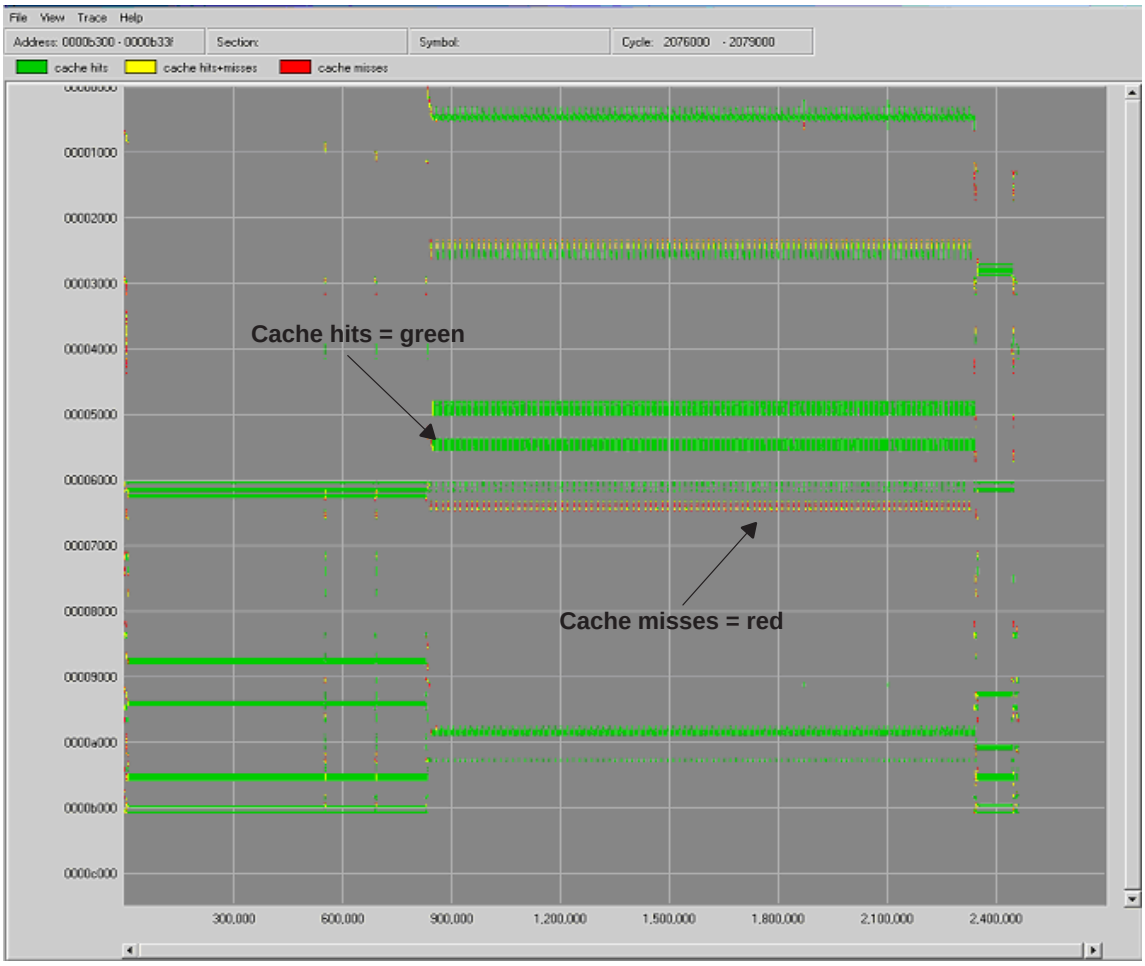

Figure 1. A First Look at Cache Analysis

This section introduces cache analysis trace files and discusses the recommended workflow of using the tool to increase the cache performance. Refer to the *Cache Analysis User's Guide* (SPRU575) for a detailed description of available features and trace files.

4.1 Trace Files

Specially instrumented simulators in Code Composer Studio 2.2 have the capability to dump the cache trace events (hits and misses). The memory reference patterns as well as the symbolic information are captured by the simulator and are written to trace files which track different aspects of cache behavior: Current simulators that support cache analysis trace generation on C55x targets is the C55x cache simulator. Further detailed descriptions of these simulators can be found in the *TMS320VC55x Instruction Set Simulator Technical Overview* (SPRU599).

The type of trace files and supported events by the C55x cache simulator are described below:

- Instruction memory references (i.tip)

  This correlates instruction memory instruction cache (C55x). The supported events are cache hit and cache miss.

4.2 Cache Analysis in the Application Development Life Cycle

The cache analysis tool supports the application software development life cycle by addressing aspects of efficiency analysis targeting better memory management in two distinct stages:

- Perform efficiency analysis at the algorithm level with a particular memory context.
- Perform efficiency analysis targeted at the whole-application level when CPU and memory cycles are measured after the integration of the algorithm into an application framework.

Refer to the *Analysis Tool Kit for Code Composer Studio v2.2 User's Guide* (SPRU623) for detailed descriptions about the typical application software development cycle.

Figure 2 provides a guide as to when the cache analysis tool can be utilized.

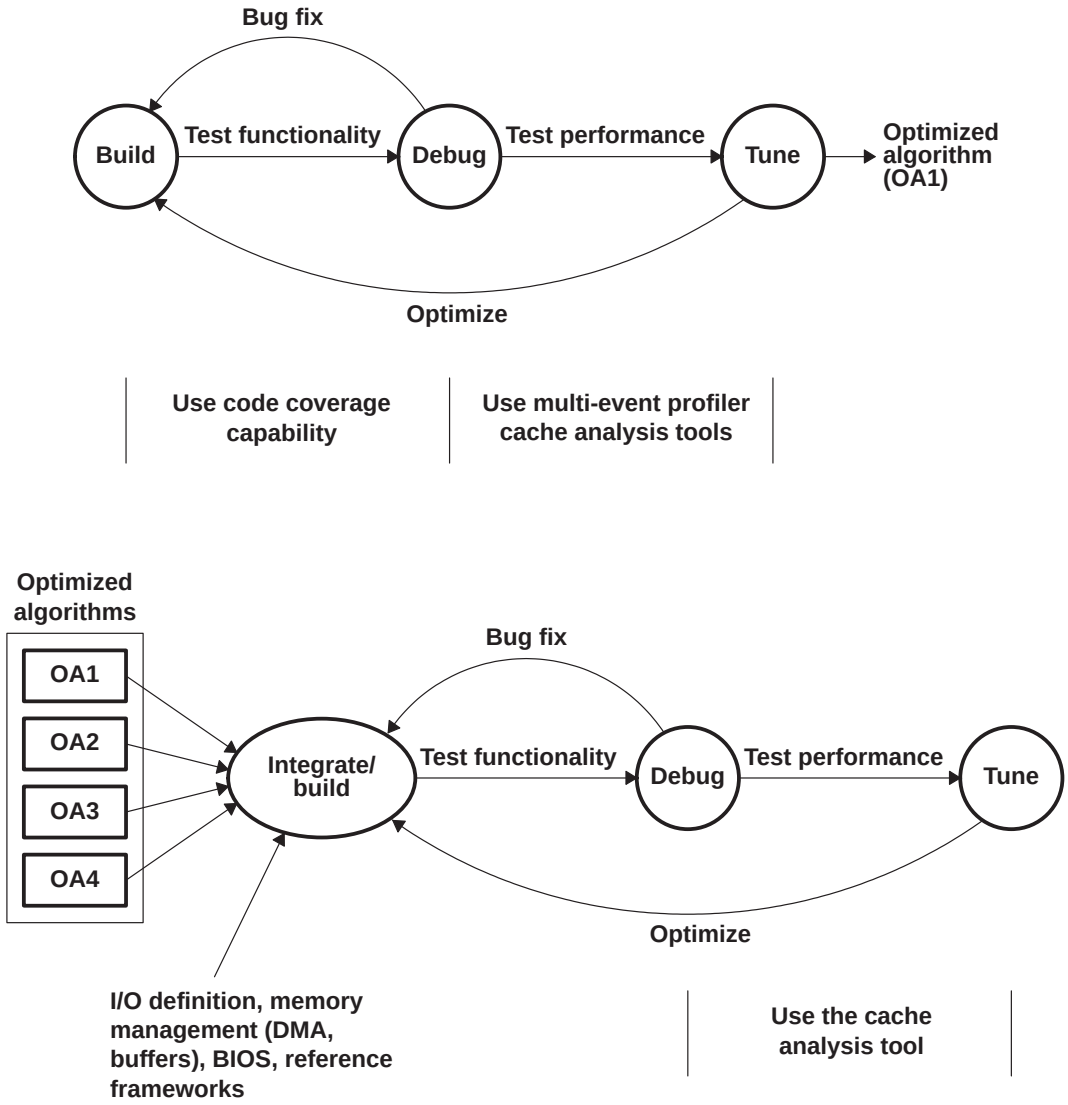

Figure 2. Using the Cache Analysis Tool

4.3 Development Flow To Increase Cache Efficiency

The recommended code development flow within the measure and optimize memory management phase involves utilizing the cache analysis tool to aid your optimization thereby increasing cache performance. The flow involves the phases described below. This application report focuses on phase 2, and will show you when to go to phase 3 for optimizing cache efficiency. We will illustrate the importance of recognizing sub-optimal cache usage areas. We will also describe how to interpret the graphical display to identify the classes of cache misses that were discussed in section 2. Based on the analysis, there are certain cache optimization techniques you can employ to improve cache performance.

The recommended flow is illustrated in Figure 3.

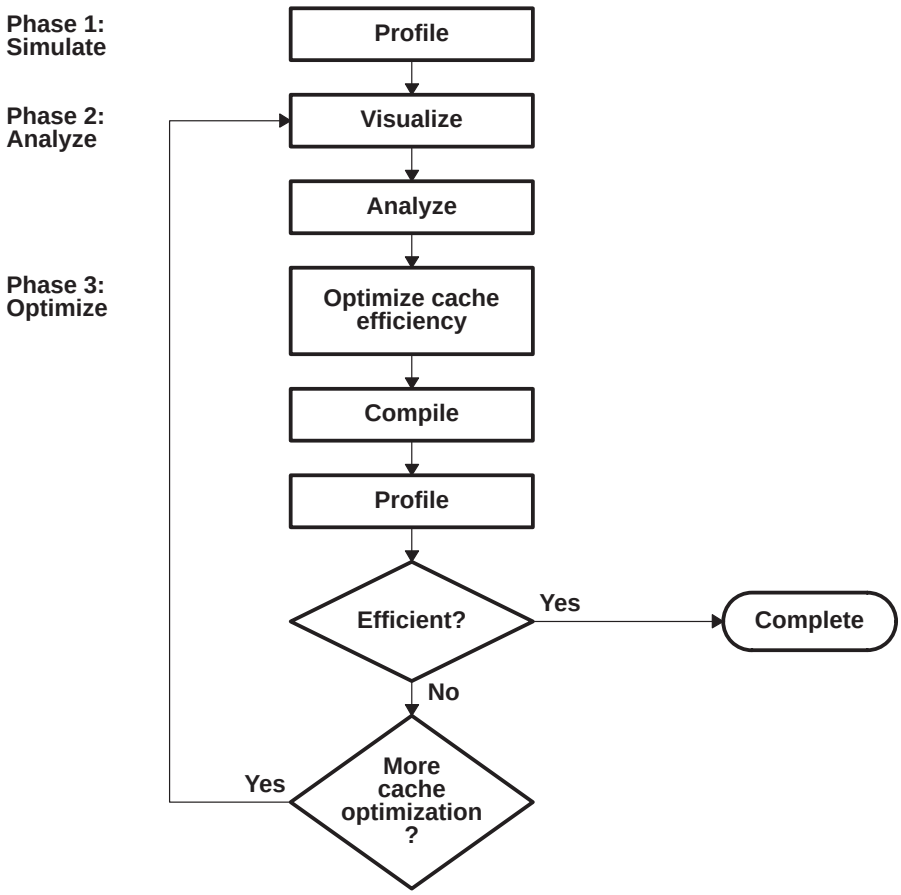

Figure 3. Development Flow to Increase Cache Efficiency

The goals for the phases in the development flow are described in Table 2.

Table 2. Development Flow

Phase	Goal
1	Generate the trace files from the cache simulators and use the simulator analysis tool to report the occurrence of cache events. To analyze the cache behavior of your code, proceed to phase 2.
2	Visualize the profile results in the cache analysis tool to identify the areas of code that are incurring cache misses. To improve the efficiency of cache, proceed to phase 3.
3	Apply optimization techniques and transformations to improve cache efficiency. Use the Simulator Analysis 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 previously described can be achieved within the C55x cache simulator environment. The cache events (hits and misses) captured by the Simulator Analysis plug-in are used as measurement for cache usage improvement. However, even though cache misses are very important characteristics to measure cache efficiency, the ultimate measure will be the effect on program execution time. Although cache simulators provide accurate cache traces, they are not cycle accurate. We need to run the application on a hardware platform such as a development board, to measure the total execution cycles for the application. This measurement should be performed before and after the code development flow described above. The setup and invocation of the cache simulators will be demonstrated in detail in section 5 with a code example. Further detail on the characteristics of the cache simulator can be found in the *TMS320VC55x Instruction Set Simulator Technical Overview* (SPRU599).

The typical user workflow reflecting the usage of the cache analysis tool is described:

1. Build and profile the application on the hardware platform
   - Obtain total CPU cycle count
2. Build and profile the application with the cache simulator
   - Use CCStudio setup to select the cache simulator
   - Enable the profile from the simulator configuration file
   - Run the application to generate trace files
3. Visualize and analyze trace data by cache analysis tool
   - Visualize the cache events
   - Locate the areas of caches misses
   - Analyze the display to identify the classes of misses
4. Optimize cache performance
   - Apply certain optimization techniques for different classes of cache misses

Steps 2−4 can be repeated as needed until the cache misses are reduced to meet particular efficiency needs. Once the developer is satisfied with the cache performance, he/she can go back to step 1 to see the improvement of the overall CPU performance on the application.

5 Using Cache Analysis Tool with an Example

This section walks you through the code development flow to increase the cache performance. A simple JPEG Encoding code example is used here to illustrate how to use the software development tools in each phase of the development flow. The complete example source code is provided with this report. The following steps will guide you through the process of measuring CPU performance on a hardware platform, generating trace files with the cache simulator, and making use of the cache analysis utility to maximize the cache performance.

Analysis toolkit needs to be installed in order to use cache analysis. The tool kit is available as an update over Code Composer Studio IDE v2.2 via the update advisor (Help →Update Advisor).

The example is demonstrated within TMS320VC5502 environment. The I-Cache implemented in the VC5502 is a 2-way set associative cache that holds up to 16K bytes. Each cache way is 8K bytes and the cache line is 64 bytes. This information is necessary to understand the cache behavior with this particular example.

5.1 About the Example

This is a program that simulates a typical Joint Photographic Experts Group (JPEG) encoding process utilizing some routines from C55x Image/Video Processing Library (IMGLIB). The encoding process consists of the following three data processing and transmission operations:

- IMG_sw_fdct_8x8: This routine performs the 2-D forward discrete cosine transform (FDCT) on a list of 8x8 image blocks and outputs corresponding 8x8 blocks of 2-D frequency components.
- IMG_jpeg_quantize: The routine quantizes the 8x8 matrix outputs from the DCT module. This step corresponds to the quantization that is performed in 2-D DCT-based compression techniques.
- IMG_jpeg_vlc: This step performs Variable Length Coding (VLC) on the outputs of the quantization routine and returns a bit stream of a JPEG baseline Huffman coding. It checks ITU–T T.81 Huffman tables and performs DC and AC coefficient coding, including run-length code and variable length code.

For detailed descriptions of these functions, refer to the *TMS320C55x Image/Video Processing Library Programmer's Reference* (SPRU037). An excerpt of the source code is listed in the following figure.

```
for (blocks = 0; blocks < BLOCKS; blocks++)
{
      /* 2-D forward discrete cosine transform (DCT) for 8*8 image block */
     IMG_sw_fdct_8x8(&imageBlock[64*blocks],interBuffer);

      /* determine type */
      temp = blocks % 6;
      if (temp==0 | temp==1 | temp==2 | temp==3 )
          type = 0;
      else if (temp==4)
          type = 1;
      else // (temp==5)
         type = 2;

      /* Matrix quantization */
      if (type == 0)
       IMG_jpeg_quantize(&imageBlock[64*blocks], zigzag, y_quant_tab, Q_dct_coef);
      else
       IMG_jpeg_quantize(&imageBlock[64*blocks], zigzag, uv_quant_tab, Q_dct_coef);

      /* variable length coding */
       IMG_jpeg_vlc(Q_dct_coef, &output[blocks*128], VLC_status, type);

}
```

Figure 4. JPEG Encoding Process

Furthermore, by a relative placement in the linker command file, the three functions are placed into the memory map such that they map to the same cache line in the instruction cache. Part of the linker command file is listed below. See the *TMS320C55x Assembly Language Tools User's Guide* (SPRU280) for detailed information on the linker command file.

```
MEMORY
{
      MEM0          :   org = 0010000h, len = 0002000h
      MEM1          :   org = 0012000h, len = 0002000h
      MEM2          :   org = 0014000h, len = 0002000h
      ...
}


SECTIONS
{
      /* Functions are allocated at the memory that are conflicted in cache */
      .fdct          > MEM0
      {
         -lIMIGLIB_JPEG.lib<swdct.obj> (.text)
      }


      .quantize      > MEM1
      {
         -lIMIGLIB_JPEG.lib<JPEG_qtal.obj> (.text)
      }


      .vlc           > MEM2
      {
         -lIMIGLIB_JPEG.lib<jpeg_vlc.obj> (.text)
      }
      ...
}
```

Figure 5. Linker Command File

5.2 Walking through the examples

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 (i.e., you know how to set up it up as well as to how create and load a project). Some simplifications are used to keep the content manageable in length. Refer to the *Cache Analysis User's Guide* (SPRU575) for detailed descriptions on the features and discussion on cache optimization techniques.

5.2.1 Configuration Setup and Data Collection

Step A: Setup

A C5502 DSP test board with external memory (SDRAM) is used this application report to measure the CPU cycle counts for the example. As previously discussed, cycle accurate CPU execution time is necessary to measure overall application performance. We will assume you are familiar with the CCStudio setup interface.

The C5502 CPU is configured to operate at 300 MHz. The External Memory Interface (EMIF) module clock is derived internally from the CPU clock. As the CPU is running at 300 MHz, a divider value of 4 should be used to generate the EMIF clock frequency from the CPU clock. The external SDRAM thus operate at 75 MHz (300MHz/4).

1. Start the CCStudio setup utility.
2. Select proper system configuration for the target board.
3. Save the configuration and exit the setup utility.

Step B: Compile and build JPEG_Encoding.pjt project

1. Start Code Composer Studio.
2. Open the JPEG_Encoding project that is provided with this application report.
3. Compile and load the program.

Step C: Profile

We will use Code Composer Studio Profiler to gather the total CPU executed cycles. Please refer to Code Composer Studio online help for further detailed information on the plug-in.

1. Choose menu item Profiler → Enable clock to enable the profile clock. A check mark is displayed besides this menu item when the clock is enabled.
2. Choose menu item Profiler → View Clock. The clock window appears and displays the total CPU executed cycles.
3. Run the application to completion. Notice that the total CPU cycle count is 303,723 cycles.

Step D: Set up the cache simulator

This step is to select the appropriate configuration for the C55x cache simulator and set up the simulator configuration file for data connection. See the *Analysis Tool Kit for Code Composer Studio v2.2 User's Guide* (SPRU623) for more details.

1. Launch setup and select C55x cache simulator from the standard configurations. The configuration will be graphically displayed in the System Configuration pane in the form of C55x Cache Simulator under My System icon.
2. Right-click on the C55x Cache Simulator and select Properties from the pop-up menu. This opens the Board Properties dialog box.
3. Select the Board Property tab, and click on the file browse button labeled ".." to the right of the configuration file (see Figure 6). Browse and choose configuration file SIM55xx_csim_profile.cfg. This file is supplied with the analysis toolkit to provide a convenience way of enabling data collection.
4. Open the file with any text editor and look for the statement CACHE_PROFILE 250 under the MODULE PROFILE section. This directive turns on cache analysis trace generation and specifies the collection time interval (in cycles). The default for C55x cache simulator is set as 250 cycles.
5. Change the sample interval from 250 to 10 to have a larger resolution for demonstration purposes. Save the changes and close the editor.

NOTE: It is suggested that you make separate copies of these configuration files if you need to modify the default settings for cache analysis.

6. Select the Startup Gel File(s) tab and click on Finish button. Save the changes to system configuration and exit setup.

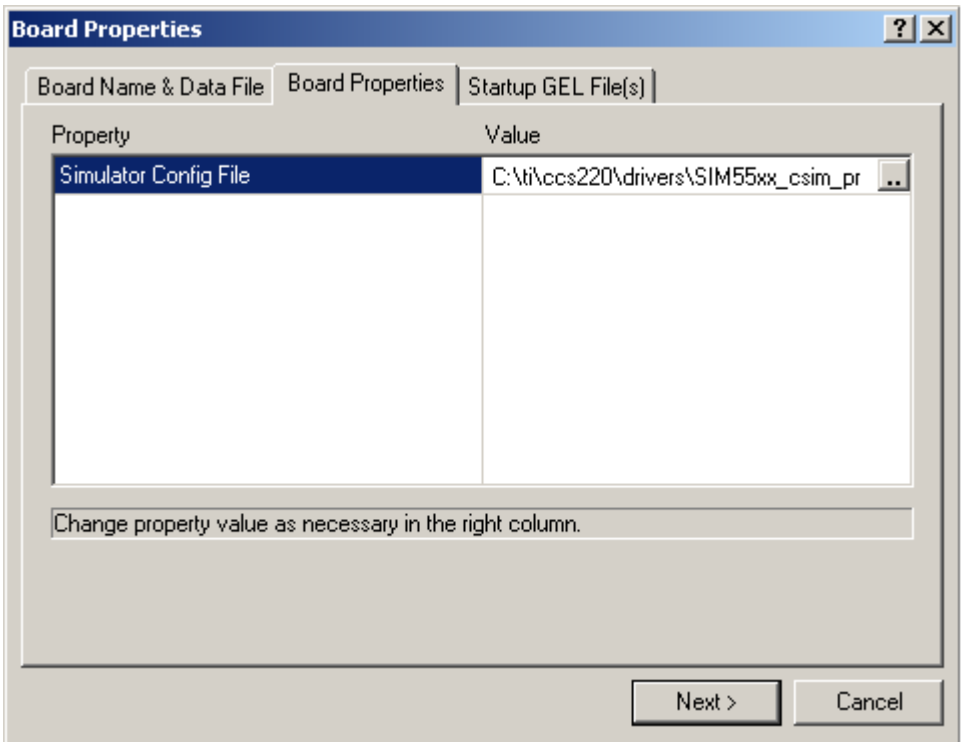

Figure 6. Board Properties Window

Step E: Run the application to generate trace file and gather cache statistics

1. Launch Code Composer Studio and open the JPEG_Encoding.pjt.
2. Open source file JPEG_Encoding.c, locate the routines that enable and disable I-Cache, which are located before and after the *for* loop in the main function. Comment them out as C55x Cache Simulator automatically cached all the program memory accesses.
3. Build the project and load the program.

   We will use Simulator Analysis plug-in to gather cache statistics. Please refer to Code Composer Studio online help for further detailed information on the plug-in.
4. Choose menu item Tools → Simulator Analysis to invoke the Simulator Analysis plug-in. Simulator Analysis reports the particular system events that allow us to accurately monitor and measure the performance of our program.
5. In the Analysis Setup window, enable the analysis and set the events to count. The events we are interested in are instruction cache events. Select the events for I-Cache hits and misses, which will be displayed in Simulator Analysis Window.
6. Run the application to completion. When the application executable is loaded in step 3, data collection starts implicitly. While the program runs, the trace file is generated into the directory from which the program was loaded by default. The data collection is automatically stopped when the program execution completes.

NOTE: Application execution is said to be complete if the application reaches the special symbol “C$$EXIT”. This symbol is reached when the application execution returns from *main( )*. The symbol is part of the *RTS* (run time support*)* library.

Once the program terminates, the statistics of selected events will be displayed in the Simulator Analysis Window. Figure 7 is a screen shot of this window.

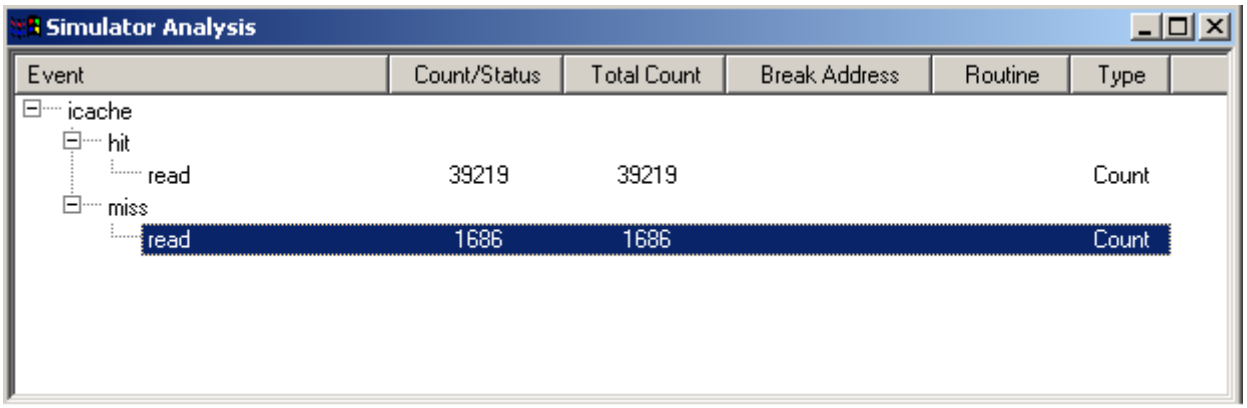

Figure 7. Simulator Analysis Window

Notice there are 1,686 instruction cache misses.

7. Now the trace data have been collected and we are ready to launch the cache analysis tool. Choose Tools → Analysis Toolkit → Cache Analysis to launch cache analysis tool. By default, it opens the i.tip file.

For details regarding to data collection, see the *Analysis Tool Kit for Code Composer Studio v2.2 User's Guide* (SPRU623).

5.2.2 Instruction Cache (I-Cache) Analysis and Optimization

Step A: Visualize and analyze the i.tip file

1. Open the i.tip file to visualize the program memory access in I-Cache.
2. Select menu item View → Cycle resolution and adjust it to 20. This displays all the program references and provides an overview of cache access pattern and points out the *hotspots* where most caches misses occurs.
3. From the trace menu, un-check cache hits events to view the cache misses only. The graph will look similar to Figure 8.

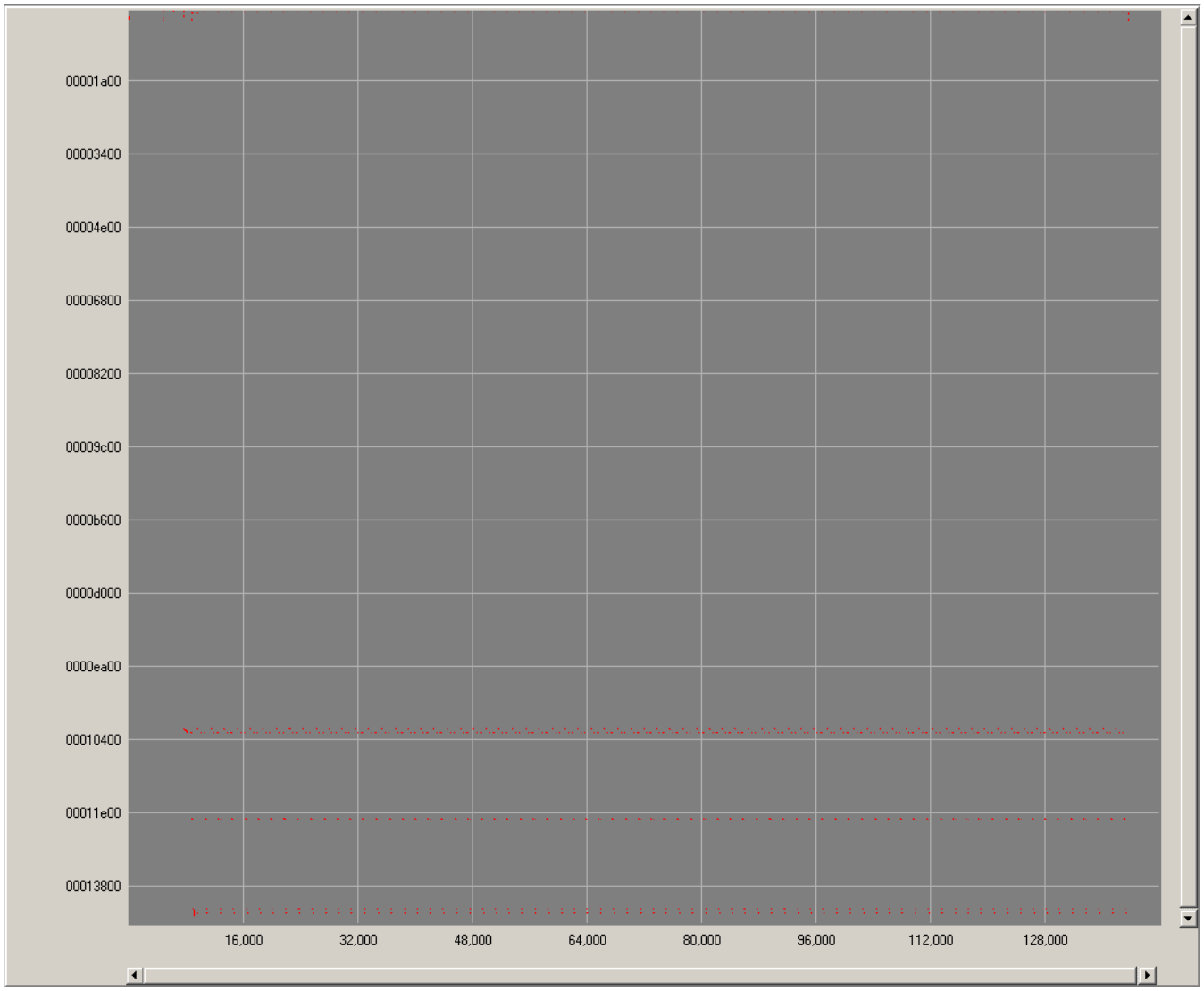

Figure 8. Cache Misses in I-Cache

Notice there are three horizontal red bars located at the lower portion of the image. This signifies that the same piece of code repeatedly misses the cache within a short period of time.

4. Move the cursor on top of areas where cache misses occur. The information area then updates with current address range, cycle range as well as section and symbol corresponding with pixel under the cursor. This provides the information of where (address) and when (cycle) cache events occur on which program (section and symbol) memory access. Notice that the functions associated with the three horizontal bars from top to bottom are IMG_sw_fdct_8x8, IMG_jpeg_quantize and IMG_jpeg_vlc respectfully.

   Recall that these three functions have been placed by the linker such that they overlap in I-Cache. When they are called consecutively in a loop, they replace each other from cache before the functions are called again (I-Cache is two-way set associative). Each function call causes cache misses for all iterations. These misses are known as conflict misses.

5. The interference grid feature supplied with the cache analysis tool can easily highlight this kind of miss. Select the origin of the grid at an address within the memory range of function IMG_sw_fdct_8x8 (represented by the top horizontal bar). A dark grid will be drawn at any conflicted memory address, which is every other 8K byte address (half size of I-Cache) for a two-way set associative I-Cache on C55x.

   A screen shot with the *interference grid* enabled is shown in Figure 9. Notice a dark line is drawn on top of the lower horizontal bars, that is, the conflicting functions.

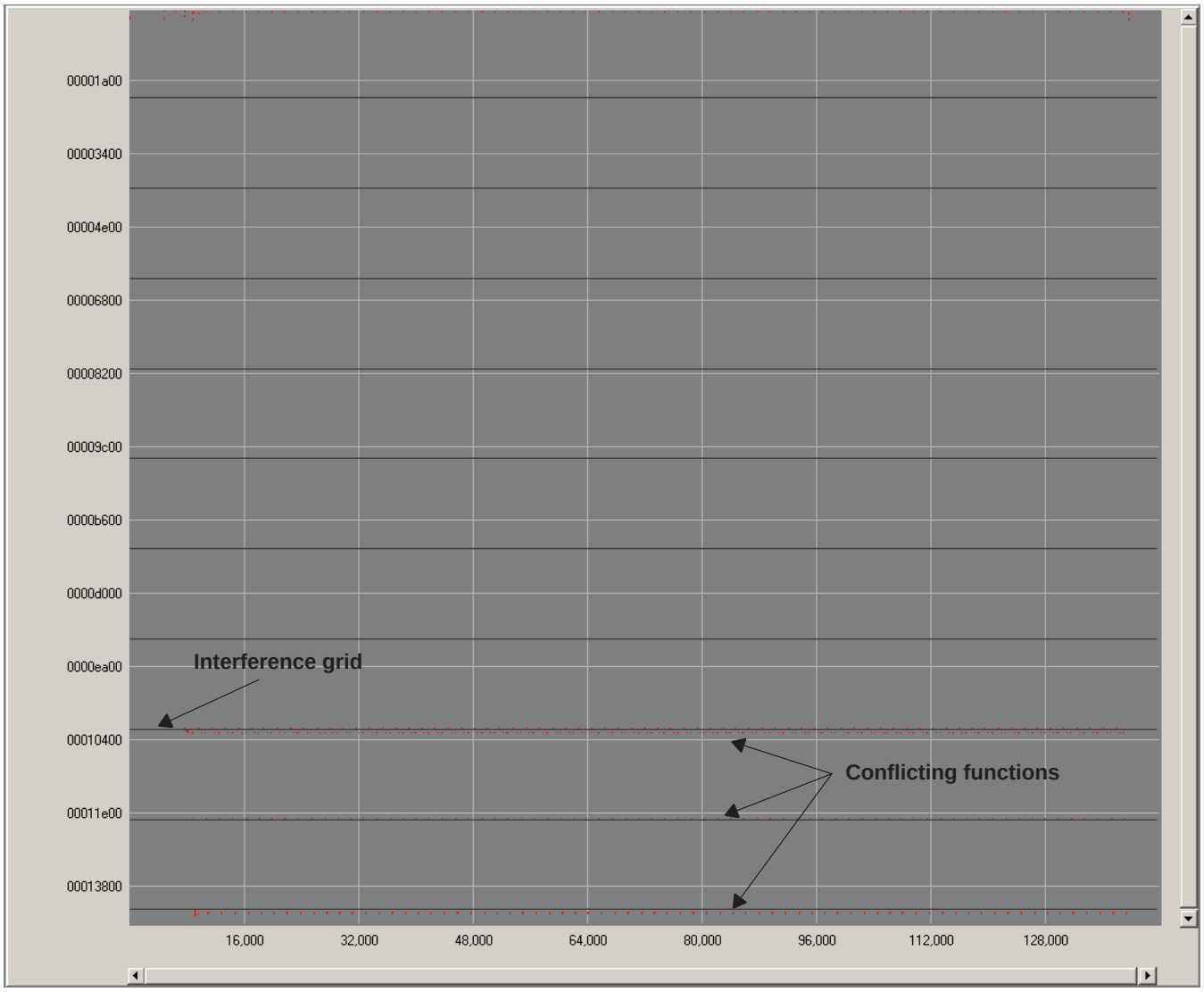

Figure 9. i.tip File with Interference Grid Enabled

Step B: Eliminating the Conflict Misses

Knowing the cache misses are conflict misses due to three functions mapped to the same cache lines, we can place the affected functions in different memory locations so that they do not overlap in cache. One possible solution is to allocate the functions contiguously in memory. This can be accomplished by editing the linker command file.

1. Modify the linker command file supplied with the original source with the changes shown in Figure 10. The source file can be found in lnk2.cmd file.

```
SECTIONS
{
     /* Functions are allocated at the memory that are conflicted in cache */
     .fdct           > MEM0
     {
                   -lIMIGLIB_JPEG.lib<swdct.obj> (.text)
     }
     .quantize       > MEM0
     {
                   -lIMIGLIB_JPEG.lib<JPEG_qtal.obj> (.text)
     }


     .vlc            > MEM0
     {
                   -lIMIGLIB_JPEG.lib<jpeg_vlc.obj> (.text)
     }
...
```

Figure 10. Modified Linker Command File

2. Re-link the application with the new function placement.
3. Load the program. The counts of cache events in Simulator Analysis window will be automatically reset to zero every time you load a program.

Step C: Measuring the Improvement

1. Repeat the profile steps illustrated in sections 5.2.1 and 5.2.2 to generate trace and gather cache events data.

   Table 3 gives a comparison of L1P events before and after the optimization. The conflict misses between the two functions are remedied.

Table 3. L1P Events Comparison

L1 P Events	Before Optimization	After Optimization
hit. Read	39,219	40,855
miss. Read	1,686	122

2. Open the new instruction trace file (i.tip) from the cache analysis tool.
3. Use the zoom-in feature (View → Zoom in) to mark a rectangular area with respect to referencing the three functions discussed earlier. This gives a much detailed view of the memory access.

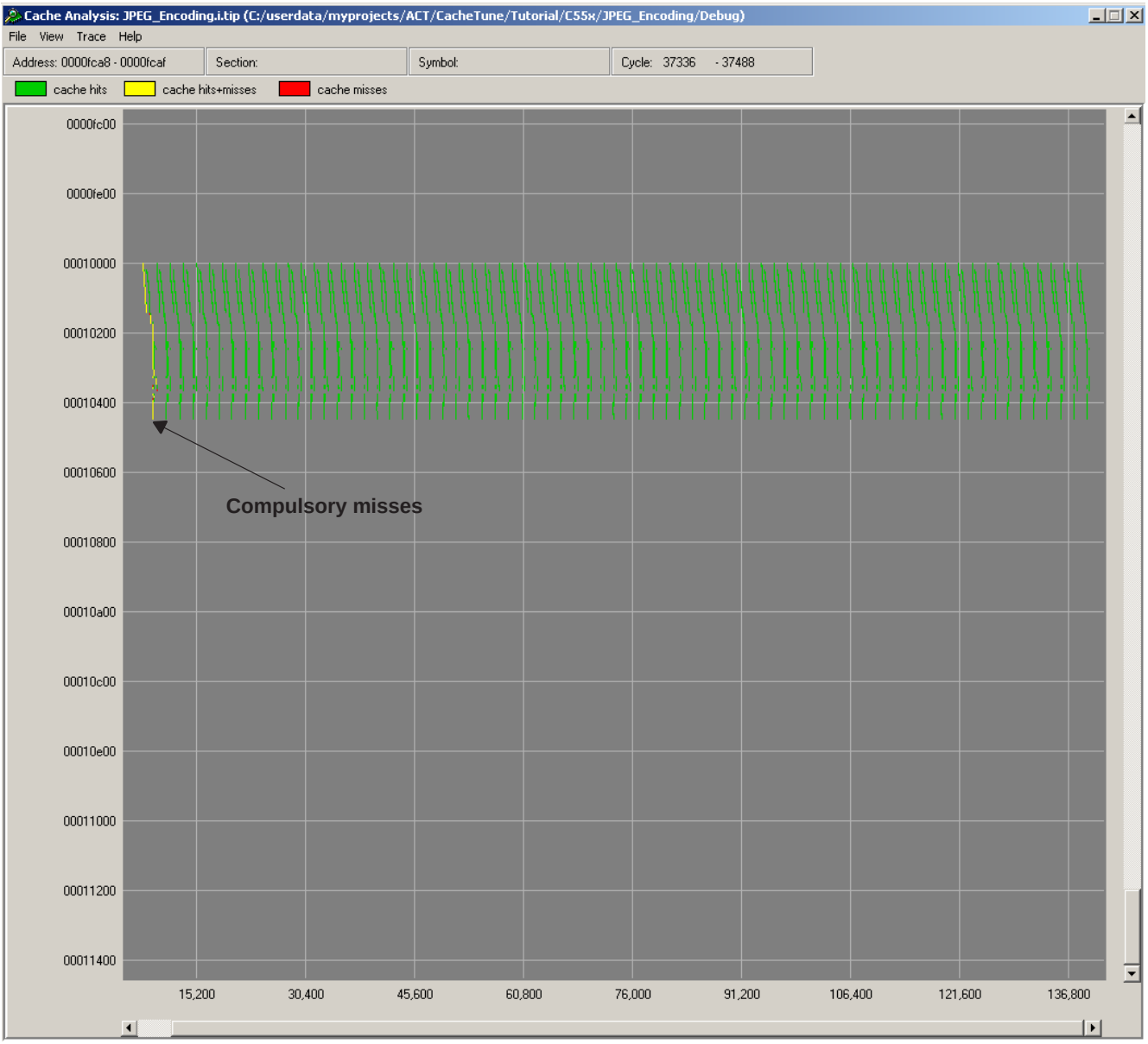

Figure 11. i.tip File After Optimization

From the display, notice the three functions are now grouped together and the conflict misses are eliminated. The cache misses now are mainly compulsory misses due to firstly accesses to the program.

5.2.3 Overall System Improvement

After we are satisfied with the cache performance from a miss perspective, we still need to go back to the device simulator to get the accurate CPU cycle count to measure the overall application performance.

Repeat the profile steps within the test board; the new cycle count is now 190,457 cycles. This is a 37% improvement compared to the initial program. By optimizing the cache effectiveness, the overall performance of the application is boosted significantly.

6 Conclusion

By using color-coding schemes for the hits or misses in the cache, the cache analysis tool provides a way to visualize the memory reference of a program over time to help the developer to identify the area incurring cache misses and the classes of misses, hence providing a blue print for applying optimization techniques to improve cache performance.

The example walked through the steps necessary to utilize the cache analysis tool. The tool helps us quickly identify functions conflicting with each other in the instruction cache. We easily remedy these conflict misses by reallocating the conflicting functions using the linker command file. The conflict misses in the instruction cache are eliminated completely without any extra coding effort and compilation. The overall system performance is boosted up by 37% after optimizing the cache effectiveness.

The cache analysis 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.

7 References

1. *Cache Analysis User's Guide* (SPRU575)
2. *Analysis Toolkit for Code Composer Studio v2.2 User's Guide* (SPRU623)
3. *TMS320VC5501/5502 DSP Instruction Cache Reference Guide* (SPRU630)
4. *TMS320VC5510 DSP Instruction Cache Reference Guide* (SPRU576)
5. *TMS320C55x Instruction Set Simulator Technical Overview* (SPRU599)
6. *TMS320C55x Image/Video Processing Library Programmer's Reference* (SPRU037)
7. *TMS320C55x Assembly Language Tools User's Guide* (SPRU280)
8. David A. Patterson and John L. Hennessy, *Computer Organization & Design*, Second edition, Morgan Kaufmann, 1998

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