Cache Analysis for Code Composer Studio ™ v2.3 User's Guide

SPRU575C
April 2004

Printed on Recycled Paper

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

Preface

Read This First

About This Manual

This manual provides information regarding the Cache Analysis plugin for Code Composer Studio.

How to Use this Manual

This book is devided into five chapters which are described here:

Chapter 1 introduces the cache analysis plugin and gives a quick tour of how **it works.**

Chapter 2 discusses the feaures of the cache analysis display which consist of menus, a graphical display area, and an information display area located below the menu.

Chapter 3 includes information about the trace files which contain all necessary data to visualize the memory reference pattern of an application.

Chapter 4 reviews the basics of cache, and provides techniques to help eliminate cache misses.

Chapter 5 provides example programs.

Notational Conventions

This document uses the following conventions.

- Program listings, program examples, and interactive displays are shown in a `special typeface` similar to a typewriter's. Examples use a **`bold version`** of the special typeface for emphasis; interactive displays use a **`bold version`** of the special typeface to distinguish commands that you enter from items that the system displays (such as prompts, command output, error messages, etc.).

  Here is a sample program listing:

```
0011  0005  0001          .field    1, 2
0012  0005  0003          .field    3, 4
0013  0005  0006          .field    6, 3
0014  0006                .even
```

The information in a caution or a warning is provided for your protection. Please read each caution and warning carefully.

Trademarks

All trademarks are the property of their respective owners.

Contents

1 Cache Analysis 1-1
1.1 Introduction 1-2
1.2 Work Flow 1-3
1.3 Quick Tour 1-4
1.4 Generating Executables 1-18
1.5 Generating Trace Files 1-19

2 Cache Analysis Display Features 2-1
2.1 Window Title Bar 2-2
2.2 Menu Bar 2-2
2.3 Trace Menu 2-5
2.4 Informational Display Area 2-7
2.5 Legend Area 2-8
2.6 Horizontal Axis 2-8
2.7 Vertical Axis 2-8
2.8 Graphical Display Area 2-8

3 Trace Files 3-1
3.1 Trace File Contents 3-2
3.2 Instruction Memory References (.i.tip) 3-3
3.3 Data Memory References (.d.tip) 3-3
3.4 Instruction Memory Cross Reference (.x.tip) 3-4

4 Cache Overview 4-1
4.1 Basic Operation 4-2
4.2 Set Associativity 4-4
4.3 Copy Back vs. Write Through 4-5
4.4 Write-Allocate vs. No Write-Allocate 4-5
4.5 Cache Misses: The Three Cs 4-6
4.6 Techniques to Remedy Cache Misses 4-7

5 Example Programs 5-1
5.1 Matrix Multiply 5-2
5.2 Conflict Example 5-4
5.3 JPEG 5-5
5.4 Matrix Transpose Operation 5-6
5.5 Vector Operation 5-7

Chapter 1

Cache Analysis

Cache Analysis is a Code Composer Studio plugin that provides graphical visualization of memory reference patterns for your program over a set amount of time. This tool can help you improve the cache performance of your program.

Topic **Page**

1.1 Introduction 1-2
1.2 Work Flow 1-3
1.3 Quick Tour 1-4
1.4 Generating Executables 1-18
1.5 Generating Trace Files 1-19

1.1 Introduction

Cache Analysis is a tool that graphically visualizes the memory reference pattern of a program over time. Using color-coding, the references are categorized according to whether they hit or miss in the cache. Symbolic information, when available in the program executable, allows for easy identification of functions and major data structures. This identification enables the programmer to quickly target the areas of code and data that is incurring cache misses, and apply optimizations and/or transformations to improve cache performace.

For instance, the display can easily highlight conflict misses in the instruction cache that occur because two or more functions map to the same place in the cache (see Section 4.5). Knowing this, the programmer can modify the linker command file to allocate the affected functions so that their footprints in the cache do not overlap, thus eliminating these “useless” cache misses.

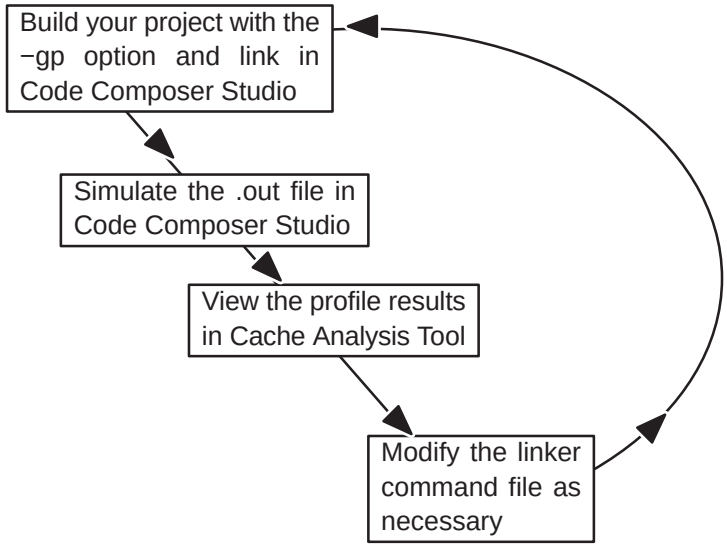
The memory reference patterns for the data and instruction caches are captured by a specially instrumented simulator and written to three trace files: <file>.i.tip,<file>.d.tip, and <file>.x.tip, where <file> is the program executable name without the extension.

Each trace file tracks a different aspect of the cache memory system behavior. The .i.tip file contains trace data that correlates instruction references with their outcomes (hit/miss) in the instruction cache. The .x.tip correlates instruction memory references to the cache outcome (hit/miss) in the data cache of loads and stores. Lastly, the .d.tip correlates data references with their outcomes (hit/miss) in the data cache. The trace files are described in more detail in Chapter 3.

The cache analysis data is available from C641x, DM642, C671x, C6211, C5510, and C5502 device simulators.

The C6000 simulators that support cache analysis, dump all the three trace files. Here, the i.tip file provides the trace data that correlates instruction references with their outcomes in the level 1 instruction cache (L1P). The x.tip file provides the trace data that correlates instruction memory references to cache outcomes in the level 1 data cache (L1D) for loads and stores. The d.tip file correlates data references with their outcomes in the L1D data cache. In the case of C5510 and C5502 simulators, only i.tip is dumped. This file has the trace data that correlates instruction references with their outcomes in the instruction cache (ICache).

1.2 Work Flow


Build your project with the −gp option and link in Code Composer Studio
Simulate the .out file in Code Composer Studio
View the profile results in Cache Analysis Tool
Modify the linker command file as necessary

1.3 Quick Tour

If you want to use your own code rather than the example code provided with Cache Analysis, browse to the directory in which your Cache Analysis traces are located (rather than the Cache Analysis example directory) when instructed. For generating Cache Analysis traces see section 1.5.

Cache Analysis only requires you to build your project with the −gp option. You must then, however, regenerate any .out files to be used with Cache Analysis to include the −gp build option (*this includes the C run time libraries (rts)*). The −gp option enables symbolic information to be displayed and includes size information about data and code objects.

The following procedures assume that Code Composer Studio has been installed in C:\ti. If this is not the case, substitute your installation directory wherever C:\ti occurs in the text.

Setup Procedure

Step 1: Run Code Composer Studio Setup.

Step 2: Through the import configuration tool, select C6416 Functional Sim Ltl Endian, if you are on the C6000 platform.

Step 3: Open the board properties (right−click on the board icon and choose properties).

Step 4: Select the Board Properties tab.

The Simulator Config File property of the board properties tab shows the entry as sim6416_functional_simulator.cfg. This configuration file does not have the necessary settings for data collection.

Step 5: To set up the simulator for data collection, select the specialized configuration file, sim6416_functional_simulator_profile.cfg using the browse button in the Simulator Config File entry field.

Step 6: If you do not want to change the default CACHE_PROFILE count mentioned in the cfg file selected in Step 5, proceed to Step 9. Otherwise, open this cfg file in a text editor. Modify the value specified for CACHE_PROFILE.

See ”Text Editor Procedure” in the next page.

Step 7: Save the changes to a different file (use a different filename) and close the editor.

Step 8: Edit the Simulator Config File property of the board (Step 5) to pick up the file from Step 6 and Step 7.

Step 9: Select the Startup GEL file(s) tab and click Finish.

Step 10: Save the changes to the Code Composer Studio configurations.

Step 11: Close CCSetup and start Code Composer Studio.

Text Editor Procedure

Step 1: Open a text editor.

Step 2: Open the file sim6416_functional_simulator_profile.cfg.

This file is located in the “drivers” subdirectory of your Code Composer Studio installation directory.

Step 3: Locate the PROFILE module in the file. The following is the sample settings for PROFILE module.

```
MODULE PROFILE;

MULTI_EVENT_PROFILE_AND_COVERAGE ON;
                CACHE_PROFILE 250;
END PROFILE;
```

Now change the value (250) in CACHE_PROFILE entry to a desired value, say 500.

Step 4: Save the changes.

Step 5: Close the editor.

Note:

The cache analysis tool screen shots shown in the Quick Tour are taken for CACHE PROFILE value of 500.

Code Composer Studio Procedure

Step 1: Start Code Composer Studio.

Step 2: Load the program <CCS−installation−directory>\examples\sim64xx\cache_analysis\jpeg\jpeg1.out.

Step 3: Run the program.

The steps above provide instructions for the Basic Usage Mode for ATK. For more information on the ATK usage modes, see the *Analysis ToolKit for Code Composer Studio v2.3 User's Guide* (SPRU623) provided with the ATK in PDF format.

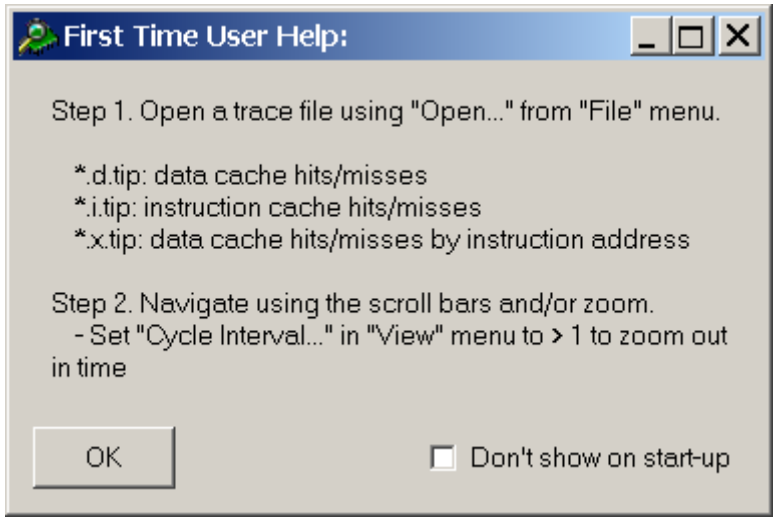
Cache Analysis Procedure

Step 1: Start Cache Analysis.

There are three ways to invoke Cache Analysis:

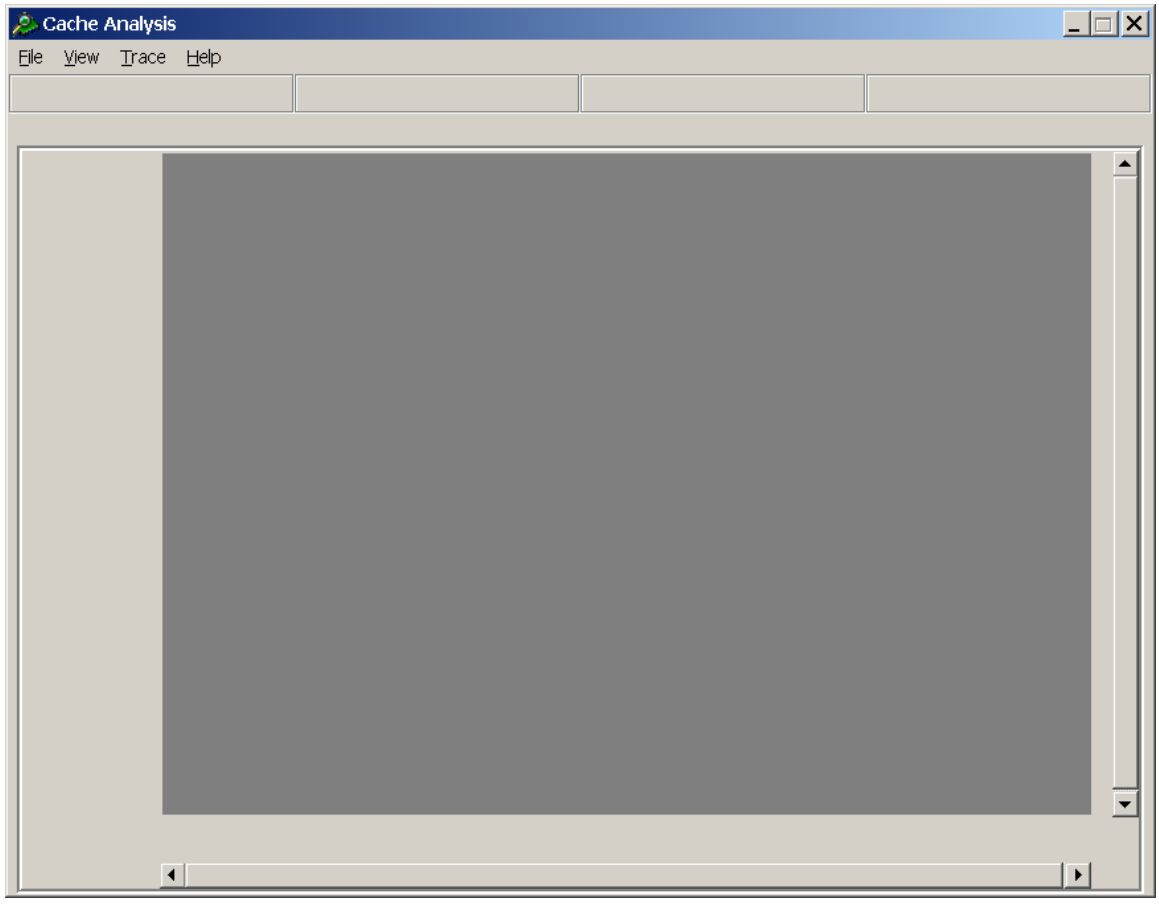
- To view data from the current execution, use the Tools→Analysis Toolkit→Cache Analysis button to open the jpeg1.i.tip file.
- Invoke Cache Analysis without any file loaded by selecting Start Menu→Program→Texas Instruments→Code Composer Studio 2 (C6000)→Analysis Toolkit→Cache Analysis
- Run the actual executable from the command line in any shell: (<CCS−installation−directory>\bin\utilities\CacheAnalysis\cacheanalysis.exe)

The First Time User Help window is displayed.



Step 2: Click OK.

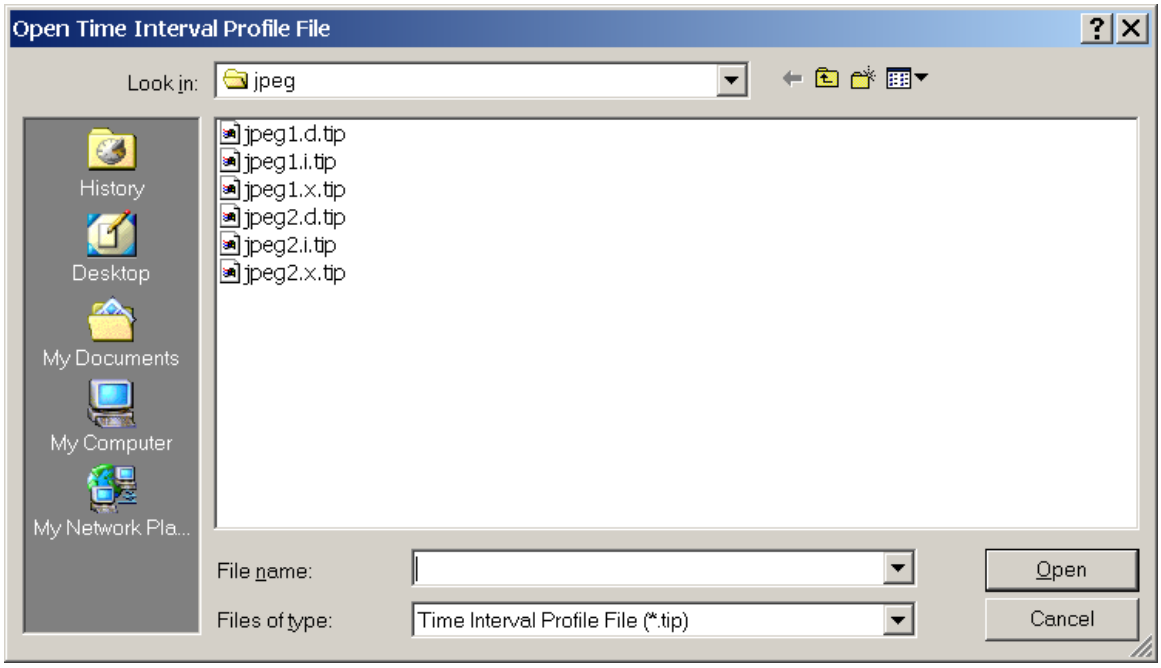
The main Cache Analysis window displays as shown below:



Step 3: Select File→Open…

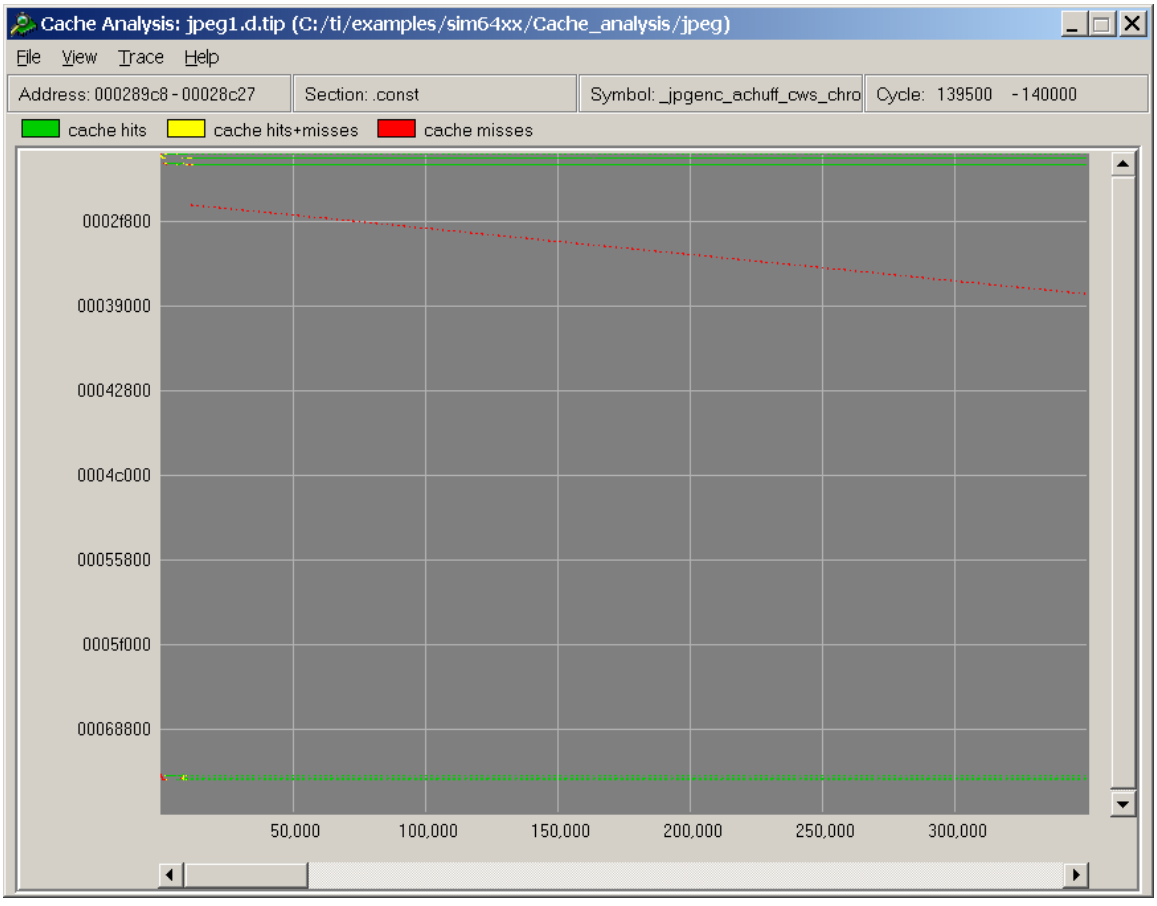
Step 4: Double click on the jpeg subdirectory (C:\ti\examples\sim64xx\cache_analysis\jpeg).

Step 5: Select the jpeg1.d.tip trace file.



Step 6: The display will be similar to the image below.

Only a small part of the trace is shown (indicated by the horizontal scroll bar). At this point, the vertical axis contains the full range of memory that was referenced in the instruction cache. The horizontal axis contains the first 400,000 cycles worth of references or so, depending on the screen resolution.



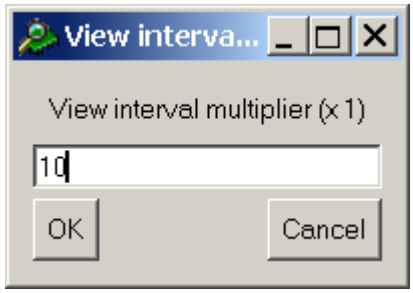
Step 7: Move the cursor across the display.

You will see the informational display area (just below the menu bar) updates with the current address range, section, symbol (if any) and cycle range corresponding to the pixel currently under the cursor.

The entire program takes over 3 million cycles to simulate.

Step 8: To view the entire trace:

View→Cycle resolution, set it to 10, and click OK. (A larger value might be required if the screen resolution is low).



Step 9: The display should now be similar to the image below.

The display shows three basic patterns. The first part of the trace corresponds to the reading of the input files. Then the actual encode is performed, followed by the writing of the output file.

Step 10: Using the Trace menu, switch between the two different color schemes: “hits/misses” and “loads/stores”.

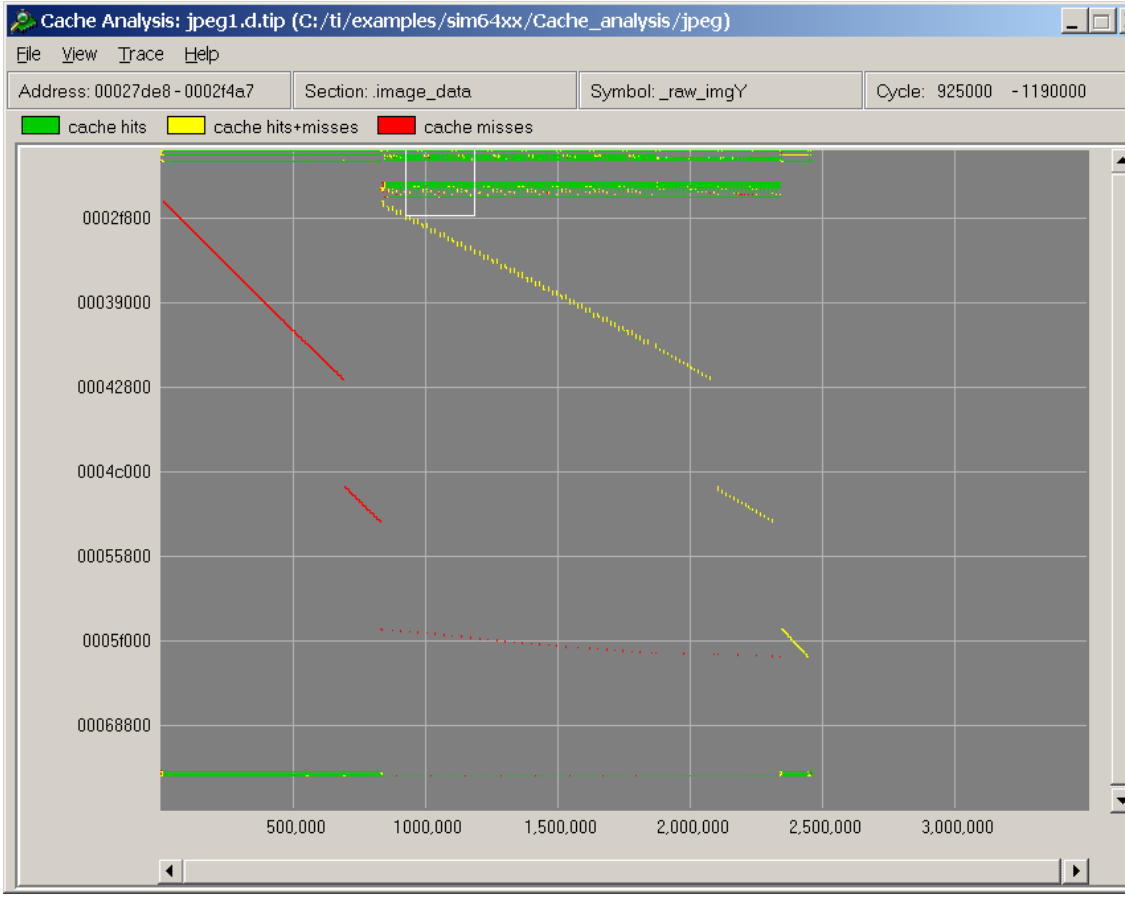
One scheme color codes the memory references according to whether they were cache hits or cache misses, the other color scheme according to whether the references were loads or stores.

Step 11: Using the Trace menu, toggle some of the references on and off.

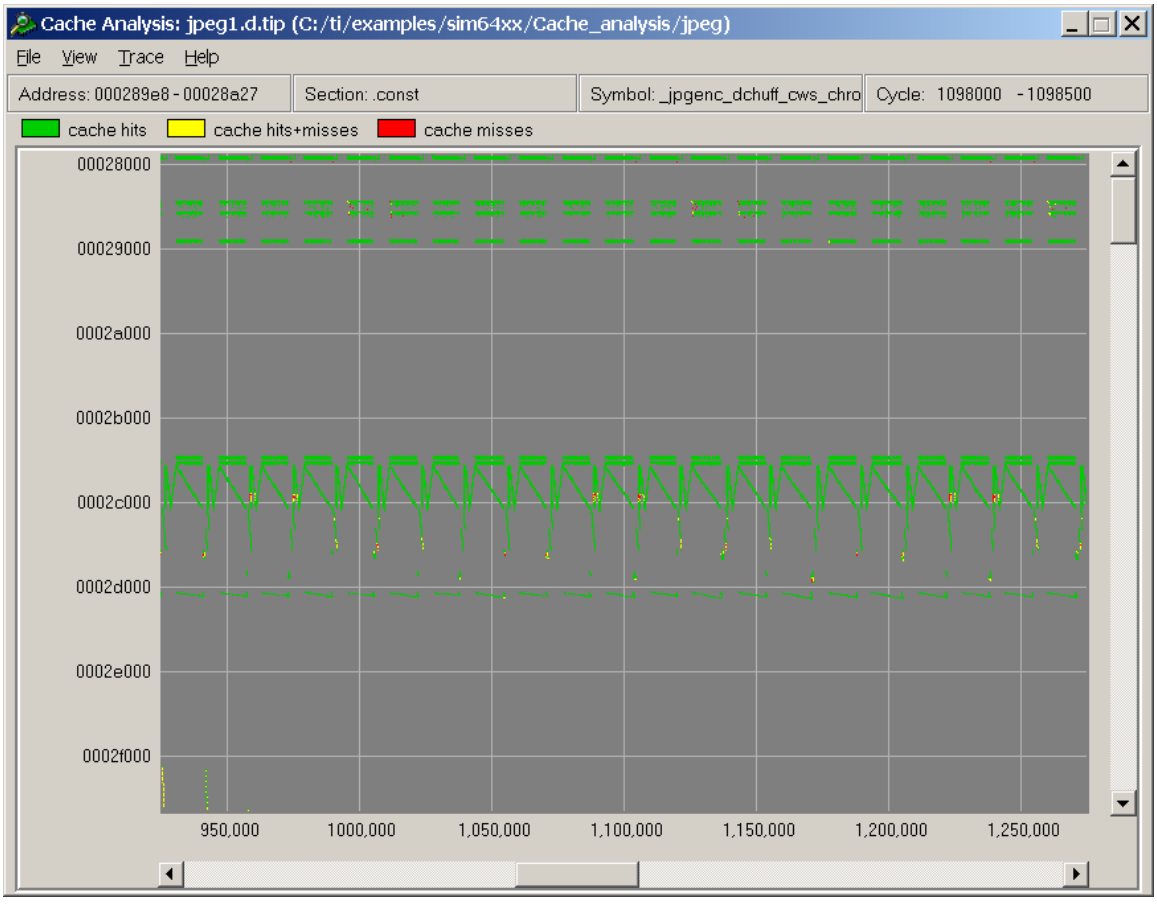
For instance, toggle “stores” and “cache hits” off to see only the loads that missed in the cache.

Step 12: Reset the color scheme to “hits/misses” and toggle all references on.

Step 13: Select View→Zoom in, and mark the rectangle shown in the partial screen shot by clicking in opposite corners.



Step 14: The resulting display should resemble the following image, showing a much more detailed picture of the memory access pattern.

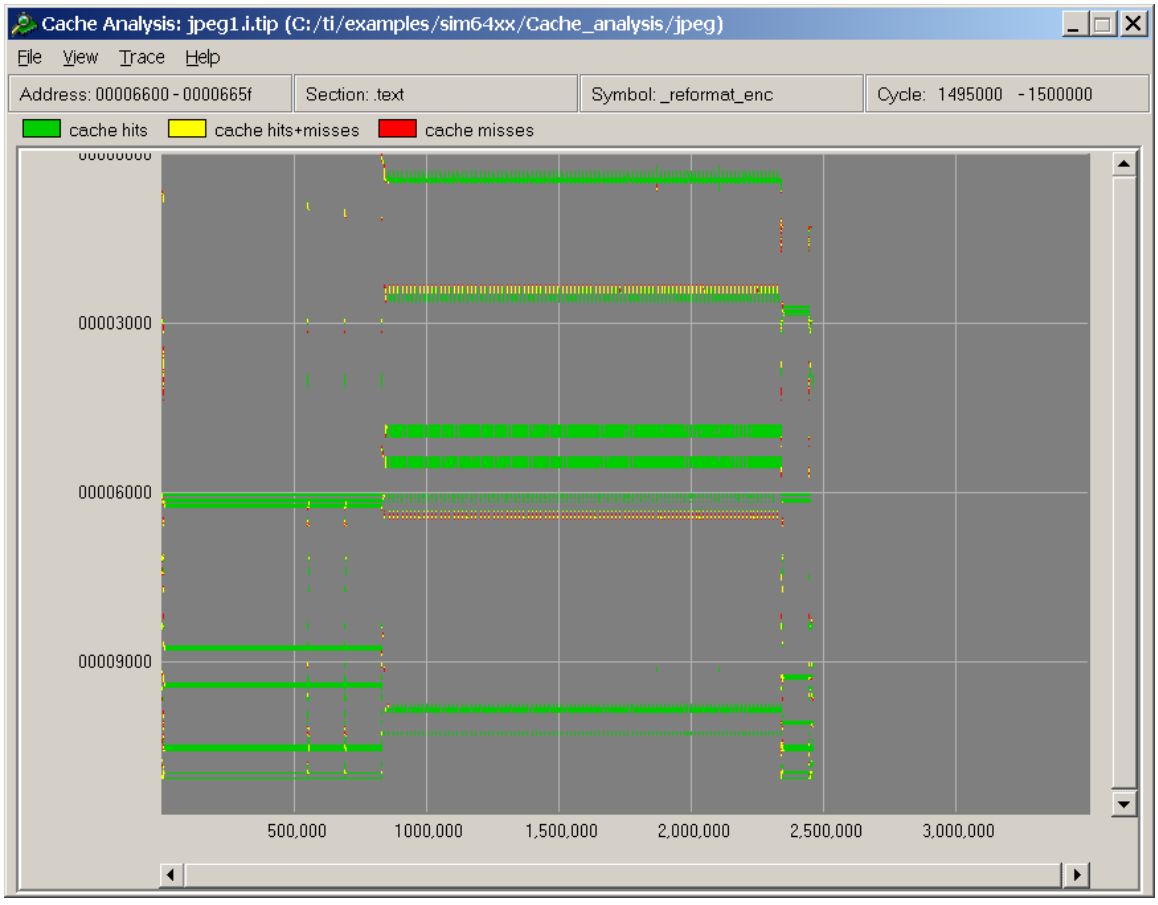


Step 15: Close this trace file.

Step 16: Open jpeg1.i.tip.

Step 17: Set Cycle resolution to 10.

Your graph should look similar to the image below.



Note again how the function names change in the Symbol display as you move the cursor up and down over the graphical display area.

Notice the horizontal bars across the display. Each bar represents a piece of code that is executed repeatedly at short time intervals over a long period of time. Moving the cursor over the bars will identify the functions containing the code, though, look out for any “overshoot”.

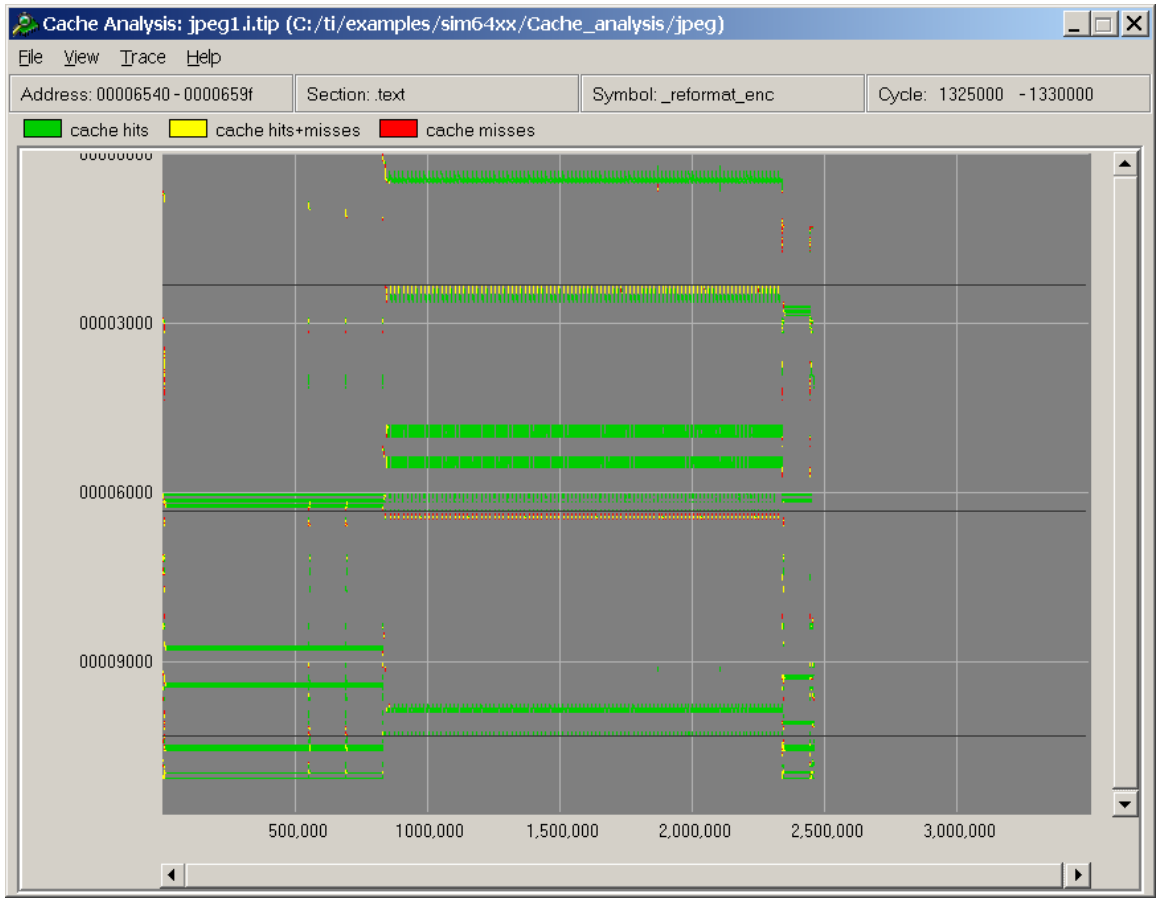
Notice also that a number of the bars are red, yellow or partly yellow along their horizontal extent. The color signifies that the same pieces of code miss repeatedly in the cache within short periods of time. These repetitive misses are due to cache conflicts.

There are two tools available for identifying which objects (functions in this case) create such conflicts in the cache. They are the Interference Grid and the Interference Shadow.

Step 18: Select View→Interference grid.

Step 19: Select the origin of the grid with a left click.

A dark line will be drawn across the display at every address that conflicts with the selected address. The following image shows the interference grid with an origin of 0x00006540.



The interference grid shows a line across the display at every address in memory that interferes in the cache with that of the origin, in this case 0x00006540. Note, the matching yellow parts of _fdct_8x8_asm (at around 0x00002540).

Step 20: To move the grid to a different origin, left click on an interference grid line and then left click at the new origin. The grid will be redrawn.

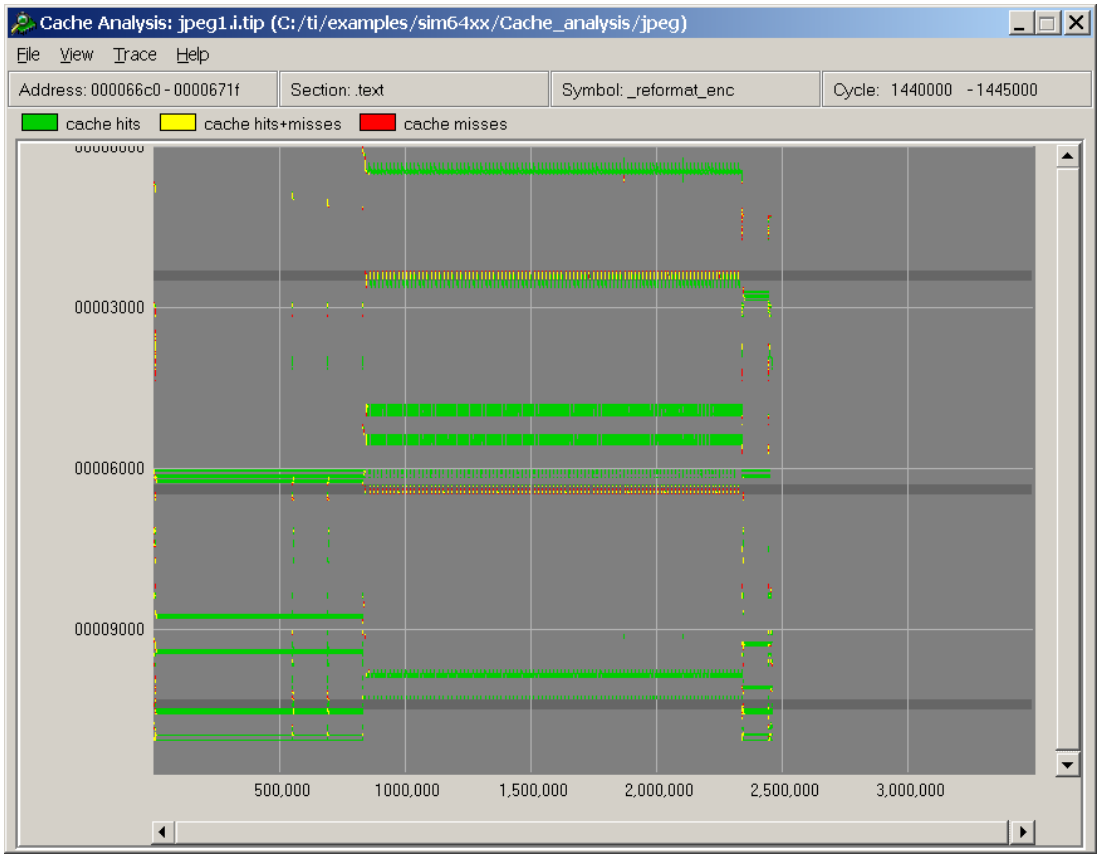
Step 21: Toggle the interference grid off again using the View menu entry.

Step 22: Select View→Interference shadow.

Step 23: Place the cursor over the function reformat_enc, located at about 0x00006540 and left-click.

The resulting display will show dark bands across the graphical display area to indicate the address ranges that interfere in the cache with this function.

reformat_enc interferes in the cache with the assembly language function _fdct_8x8_asm.



Step 24: The Interference Shadow can be moved just like the Interference Grid can be.

Step 25: The code for both the functions map to the same location in the cache. When they execute back to back, they evict each other from the cache, and, subsequently, miss the next time they execute.

One solution to eliminate the cache conflicts between the interfering functions is to group the functions that execute at the same time together in memory. Since the functions that execute when the interference occurs occupy less than 16KB of memory (the instruction cache size), allocating them contigously will eliminate any conflicts.

Grouping functions together is done by using the GROUP statement in the linker command file. However, this assumes that each function is allocated in its own subsection. For C functions, this can be achieved by using the −mo option. For linear assembly and assembly functions, using the appropriate .sect statement will put the function in a subsection of .text. For instance, for fdct_8x8_asm, using the statement:

```
.sect ".text:hand"
```

in the assembly file that defines that function will allow us to allocate that function as a separate unit.

The following shows the resulting relevant section of the linker command file after applying this approach to this example:

```
SECTIONS
{
    GROUP {
      .text:_jpgenc_ti
      .text:_reformat_enc
      .text:_encode_dc
      .text:_qrle_ac
      .text:_vlc_ac
      .text:_JPG_byte_stuff
      .text:_memcpy
      .text:hand
    }           >       PMEM
    .text       >       PMEM
```

For details regarding the linker command file, see the *Assembly Language Tools Users' Guide* (SPRU186).

Step 26: The result of applying this modification can be seen by repeating the above steps using jpeg2 program.

Step 27: Once you have tuned your code, exit cache analysis, return to Step 8 and begin the process again.

1.4 Generating Executables

Executables that are suitable for use with cache analysis are generated using the standard code generation tools for C6000 and C5500. You can use the standard Code Composer Studio installation to generate your executable file. In order to enable symbolic information to be displayed and to include size information about data and code objects the executables need to be recompiled with the –gp option. **This includes the rts libraries.**

If the size information is not available, the symbol will only be displayed for the memory range containing the start address. If the vertical axis granularity is such that the start addresses of multiple symbols share the same memory range, only one will be shown. The other symbols should become visible upon zooming in on that particular region, and thus making the memory ranges smaller.

At this point, only a very limited amount of symbolic information is supported when the executable is compiled with DWARF debug information.

1.5 Generating Trace Files

Cache analysis trace files are generated within Code Composer Studio using a specialized simulator configuration (.cfg) file. This file has a "_profile" appended in its name. It is suggested that you make separate copies of these configuration files if you need to modify the default settings for cache analysis.

The cache analysis trace generation is controlled by the following directive in the PROFILE module in the simulator configuration (.cfg) file:

- CACHE_PROFILE <N>;

  This directive turns on Cache Analysis trace generation and specifies <N> (a positive number) as the sample interval. The sample interval is the size (in cycles) of the time window over which the simulator summarizes the memory references before writing out a trace record.

Note:

Even if you select a small sample interval, you can use the "Cycle resolution…" in the "View" menu to display the trace data at an integral multiple of the sample interval.

An example of a PROFILE module that enables cache analysis tracing with a sample interval of 250 cycles.

```
MODULE PROFILE;
   CACHE_PROFILE 250;
END PROFILE;
```

Chapter 2

Cache Analysis Display Features

The cache analysis display consists of menus, a graphical display area, and an information display area located below the menu.

Topic **Page**

2.1 Window Title Bar 2-2
2.2 Menu Bar 2-2
2.3 Trace Menu 2-5
2.4 Informational Display Area 2-7
2.5 Legend Area 2-8
2.6 Horizontal Axis 2-8
2.7 Vertical Axis 2-8
2.8 Graphical Display Area 2-8

2.1 Window Title Bar

The Window Title bar shows the text cache analysis and the full path name of any opened trace file.

2.2 Menu Bar

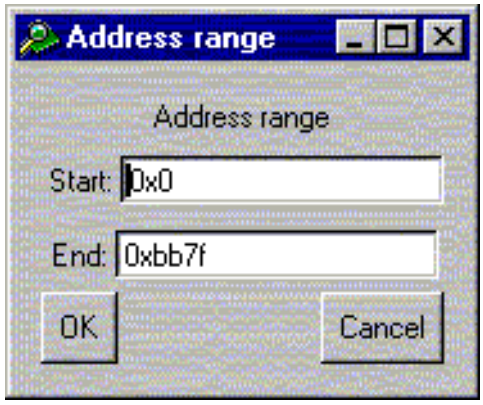
Table 2−1. File Menu

Open… **Ctrl+O**	Opens a time interval profile (.tip) trace file and displays the first screen−width of its contents.
Save as...	Saves the time interval profile (.tip) trace file under a new name.
Close… **Ctrl+W**	Closes the current .tip trace file and clears the display.
Quit Ctrl+Q	Exits the program.

Table 2−2. View Menu

Zoom in Ctrl++	Allows you to select a rectangle to zoom in on. The rectangular area is selected using the mouse by keeping its left button pressed. Once the area is decided, the button is released. Doing a left click will perform zoom in operation on the select area wheras a right click will cancel the operation. Note, the zoom will in some cases display more data than was selected. This is particularly noticeable when zooming far in.
Zoom out Ctrl+−	Zooms out to the previous view.
Interference grid Ctrl+G	Selects an address with a left click of the mouse (right click cancels), and draws a line at every address in the display at every address that interferes in the cache with the selected address.
Interference shadow Ctrl+I	Selects a symbol with a left click of the mouse (right click cancels), and draws a shadow at every address range in the display that interferes in the cache with the selected symbol. This feature requires the executable to have been compiled with the –gp option. Moreover, it only works for global symbols that have a size associated with them.
Go to… **Ctrl+T**	Opens up a dialog box in which the cycle to reposition window at is specified. 'ok' and 'cancel' buttons work in the obvious ways.
Cycle resolution… **Ctrl+Y**	Allows a 'zoom out' along the horizontal (cycle axis) by specifying an integer 'shrink' factor in the dialog box.
Set address range… **Ctrl+A**	Opens up a dialog box (seen below) which allows you to specify the minimum and maximum addresses to be displayed. Note, as for "Zoom in", in cases where small ranges are selected, more data than anticipated may be displayed. Also, you can use "Zoom out" to revert to the previous view after having used "Set address range...".

Figure 2−1. Cache Analysis Tool Address Range window



2.3 Trace Menu

The trace menu varies from one trace file type to another, but is organized in the same way across all trace types. The menu is divided into three sections separated by horizontal lines. The top section, comprising one set of options, lists the different color schemes that can be used on the current trace. For .i.tip trace files, there is only one button, but for .d.tip and .x.tip trace files there are two.

The middle section, comprising a second set of options, lists the different selections toggles that can be applied to the current trace. The set of toggles is also dependent on the trace type, as the i.tip trace files have 2 toggles, .d.tip files have 4 toggles, and .x.tip files have 5.

Note:

Toggles turn off a particular reference type, and reference types are overlapping. That is, cache hits refers to both loads and stores that hit in the cache.

Subsequently, if you toggle off all references within a single “class,” like hits *and* misses or loads *and* stores (where available), you will toggle off *all* references. This occurs because there would be no references that were loads and/or stores that didn’t either hit or miss in the cache.

Instruction cross-reference traces provide a partial exception, since there is yet one more class of references: non-memory instructions.

The last section contains two entries used to display information about the currently selected trace.

Not all the menu options listed are available in all trace files.

- **data cache hits/misses:** Selects the data cache hits/misses color scheme, where the references are color coded according to whether they contained load and store instructions that hit or missed in the cache. This and data cache loads/stores, when available, comprise a set of options.
- **data cache load/stores:** Selects the data cache hits/misses color scheme, where the references are color coded according to whether they contained load and store. This and data cache hits/misses, when available, comprise a set of options.
- **hits/misses:** Selects the cache hits/misses color scheme, where the references are color coded according to whether they hit or missed in the cache. This and loads/stores, when available, comprise a set of options.
- **loads/stores:** Selects the read/write color scheme, where the references are color coded according to whether they read from or written to memory. This and cache hits/misses comprise a set of options.

- **cache hits:** Toggles whether cache hits are displayed or not (.i.tip and .d.tip files).
- **cache misses:** Toggles whether cache misses are displayed or not (.i.tip and .d.tip files).
- **data cache hits:** Toggles whether data cache hits are displayed or not (.x.tip files only).
- **data cache misses:** Toggles whether data cache hits are displayed or not (.x.tip files only).
- **loads:** Toggles whether reads are displayed or not (.d.tip files only).
- **load instructions:** Toggles whether load instructions are displayed or not (.x.tip files only)
- **non-memory instructions:** Toggles whether non load and store instructions are displayed or not (.x.tip files only).
- **store instructions:** Toggles whether store instructions are displayed or not (.x.tip files only)
- **stores:** Toggles whether writes are displayed or not (.d.tip files only).
- **Trace information:** Shows information related to the trace file.
- **Simulation information:** Shows the date and command line of the simulator invocation used to generate the trace file.

2.4 Informational Display Area

The Informational Display Area is located directly below the menu bar. It shows information related to the pixel currently under the cursor, and is updated when the cursor moves over the display. The area is divided into four segments: Address, Section, Symbol and Cycle.

- **Address** shows the address range represented by current pixel.
- **Section** shows the name of the coff section of the executable the address range of the current pixel belongs to. If the address range contains (parts of) more than one section, the name of the section that occurs later in the coff section header table is displayed.
- **Symbol** shows which global symbol of the executable the address range of the current pixel belongs to. If the address range contains (parts of) more than one symbol, the name of the symbol that occurs later in the coff symbol table is displayed.

Note:

Not all global symbols can be displayed in this way. Only symbols that define globally visible labels, such as functions and global variables, will be displayed. Moreover, size information is only available for symbols when debugging information is included. Profile debugging (−gp) is sufficient to include size information for functions, while full debug (−g) is required to include it for data symbols. Without size information, a symbol is only displayed for the word containing its start−address.

Size information is not supported for programs compiled with dwarf debug information (−gw).

- **Cycle** shows the range of simulation cycles represented by the current pixel. This is always a multiple of the trace sample interval.

2.5 Legend Area

The legend area is located right below the informational display area. It shows the key to the current color map. The set of color maps is specific to each trace type.

2.6 Horizontal Axis

The horizontal axis is the time axis. It shows the simulated cycle counts. The scroll bar allows easy scrolling back and forth within the trace, as well as fast repositioning. The length of the slider indicates the fraction of the trace displayed.

2.7 Vertical Axis

The vertical axis shows the addresses of the memory references. The scroll bar allows easy scrolling through the address space. By default, the entire range of referenced memory is displayed when a trace file is first opened.

2.8 Graphical Display Area

The graphical display area is the main area in the center of the window. It shows a two-dimensional plot of the memory references encoded in the trace file, color coded to highlight either cache hits and misses or the type of references (loads/stores).

As mentioned in section 2.7, the initial view of memory encompasses the entire range of referenced memory. For some applications using link maps that put objects at widely spaced memory locations, this range can be very large. If the range is too large, e.g., covers multiple megabytes, the plot of memory references will show up as very narrow bands with little detail. If this happens, either use the Zoom In or Set Address Range entries in the View menu to narrow down the address range to something that shows more detail.

Chapter 3

Trace Files

The trace files contain all necessary data to visualize the memory reference pattern of an application, including any related symbolic information available from the executable.

Topic **Page**

3.1 Trace File Contents 3-2
3.2 Instruction Memory References (.i.tip) 3-3
3.3 Data Memory References (.d.tip) 3-3
3.4 Instruction Memory Cross Reference (.x.tip) 3-4

3.1 Trace File Contents

The trace data contained in the trace file is organized as a sequence of frames, one frame for each time interval. Within each interval there are zero or more records. Each record contains an address range of at least 1 word and an attribute. The attribute encodes the event associated with the address range. The events can be:

- ❏ memory load
- ❏ memory store
- ❏ cache hit
- ❏ cache miss
- ❏ no memory access.

Not all trace files encode all events. The number of occurrences of events is not encoded, though multiple events may be encoded for each address range. For instance, within an interval, a memory load that hit in the cache might have occurred on the same word as a memory load that missed in the cache. In this case, the attribute would encode the memory load as well as both a cache hit and a cache miss.

When the trace is visualized, it is very likely that the screen resolution does not support the full resolution of the trace, unless enlarged (using zoom in). This means that each pixel must correspond to multiple words and/or time intervals in the trace. The result is that multiple trace records might have to be visualized using the same pixel by combining their attributes. This combining is undone whenever the image is enlarged sufficiently.

It is important to note that the length of the time interval is set when invoking the simulator. This means that the level of detail possible along the time axis (x−axis) is fixed for any given trace file. References within a time interval cannot be resolved (in time) with respect to each other. However, it is possible to “zoom out” along the time axis using the Cycle resolution… entry in the View menu.

Note:

There is a 2GB size limitation on the trace files. Larger files are not supported by the internal file format at this time. Moreover, some operating systems do not handle larger files correctly. The size of a trace file is typically between 0.1 bytes and 1 byte per simulated cycle. The absolute upper bound is about 15 bytes per cycle, but due to the built in compression, this is extremely unlikely.

The size of the trace file grows with the number of simulated cycles, and it grows faster when the sample interval is small. If a trace file is expected to be big, it is advisable to use a larger sample interval (> 1000).

3.2 Instruction Memory References (.i.tip)

Instruction memory references are encoded in the .i.tip file (in the level 1 instruction cache − L1P for C6x1x functional and device simulators and I Cache for C5510 and C5502 simulators).

Unlike data references, the instruction cache references are performed on whole cache blocks. Therefore, the finest resolution along the address axis (y−axis) is limited to the size of the cache block in bytes (e.g., for C64xx devices this is 32 bytes; for C55x this is 16 bytes). Moreover, it is important to bear in mind that the instruction cache references are generated early in the execution pipeline.

This early generation means that there is a number of cycles between when an instruction is fetched from the cache until it is executed. The number of cycles is dependent on the pipeline, number or instructions issued per cycle and the frequency of branches, but, in the case of C64xx devices, will not be less than five cycles and in the case of C5502 and C5510 device simulators, it will not be less than 8 cycles. This fetching ahead leads to extra fetches passed taken branches.

That is, for C64xx devices, by the time a branch instruction executes and the target instructions are requested from the cache, references for up to 5 additional 32-byte blocks of instructions have already been issued to the cache and for C55x, this would be 2 additional instruction fetches. The effect of this on the trace is that the file may contain references to instructions that were never executed, but every executed instruction is contained in the trace. The most visible result of this is the “overshoot” (fetching passed the end) that can be seen at the end of each function.

3.3 Data Memory References (.d.tip)

The data memory references are encoded in the .d.tip file. The supported events are memory load, memory store, cache hit and cache miss (in the level 1 data cache − L1D). The address granularity is 4 bytes, meaning that any access to a half word or a byte is visualized as if it were to a full word. There is no “overshoot” for the data memory references.

3.4 Instruction Memory Cross Reference (.x.tip)

The .x.tip file encodes a cross−reference of instruction memory accesses with data cache outcomes. That is, the addresses are the same as for the instruction memory reference trace (.i.tip), but the events are taken from whether there were loads and stores contained within those instructions and whether those loads and stores hit or missed in the data cache (L1D). For instance, an instruction cache reference might have memory load, memory store and cache hit events encoded if it contained at least one executed load instruction, one executed store instruction and all these data memory references hit in the data cache.

The supported events are no memory access, memory load, memory store, cache hit and cache miss (in the level 1 data cache – L1D). The no memory access event is used for instruction cache references that did not contain executed loads or stores (either there were no loads or stores in among the instructions, they were not executed, or they were executed but predicated false).

The instruction memory cross−reference trace does not have “overshoot.” However, the address granularity is the size of the cache block.

Chapter 4

Cache Overview

This chapter reviews the basics of cache, as well as providing techniques to help eliminate cache misses.

Topic **Page**

4.1	Basic Operation	4-2
4.2	Set Associativity	4-4
4.3	Copy Back vs. Write Through	4-5
4.4	Write-Allocate vs. No Write -Allocate	4-5
4.5	Cache Misses: The Three Cs	4-6
4.6	Techniques to Remedy Cache Misses	4-7

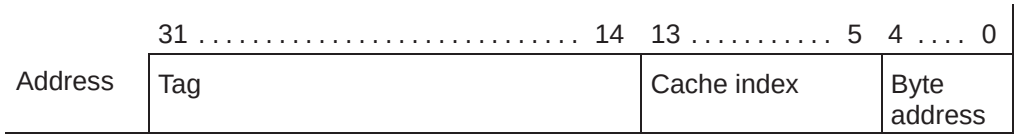
4.1 Basic Operation

A cache is a small, fast, memory used to temporarily hold data or code likely to be accessed within a reasonably short amount of time. It differs from local memory in that it is not separately addressable, but operates automatically, without programmer intervention. Caches work by taking advantage of two forms of locality seen both in instruction and data memory reference streams: temporal locality and spatial locality.

- **Temporal locality:** a memory location that has been referenced recently is likely to be referenced again.
- **Spatial locality:** a memory location that is close to a recently referenced memory location is likely to be referenced.

A cache is divided into a set of addressable units called cache blocks, that hold the data stored in the cache. The size of the cache block, always a power of 2, determines how much data is transferred in or out of the cache when new data is fetched into the cache or when data is written back to main memory from the cache. A large cache block takes better advantage of spatial locality than a small block, but may be less economical in terms of hardware and the time it takes to fill the cache block with new data.

Each cache block in the cache has a unique address. Take, for instance, a 16KB cache with 32-byte cache block size. It has 512 different cache blocks, with the addresses 0.511. This means that out of a 32-bit address, the lowest 5 bits selects the byte address within a cache block ($2^5 = 32$). The next 9 bits is used as the address of the cache block ($2^9 = 512$). This portion of the address is also called the cache index. The remaining 18 bits is known as the tag, and is stored with the cache block to uniquely identify the address range of memory it contains.



Upon a load or store instruction, the CPU sends the effective memory address to the cache. The cache controller divides the address into the byte index (lower order bits), the cache index and the tag. It then uses the cache index to select a cache block and compares the tag stored with that cache block against the tag from the CPU. This is known as a tag lookup. If the tags match, the data stored in the cache block is accessed, and using the byte index, the correct portion of the cache block is selected and sent back to the CPU. This is a cache hit. If the tags do not match it is a cache miss.

In this case, the cache controller requests a cache block worth of data, containing the data referenced by the address from the CPU, from the next level in the memory hierarchy. The data in the cache block is then replaced by the new data, and the new tag is written into the tag store. The memory access then continues as if there was a hit.

4.2 Set Associativity

The cache described in the previous section is known as a direct mapped cache. That is, each address can only be stored into a single cache block. In many cases such a cache gives very good performance while being very fast at a low cost in hardware. However, if there are two or more frequently accessed pieces of data (or code) located in memory so they map to the same cache block, references to one would cause the other to be evicted from the cache.

In the above example, if there were two variables, A and B, located within 32 byte regions 16KB (or a multiple of 16KB) apart, their cache indices would be identical. When A is referenced, B would be evicted from the cache (assuming, of course, it had been referenced first). Subsequently, if B is referenced multiple times within a short amount of time, but every reference to B is followed by a reference to A, all the references will miss in the cache. This is known as a cache conflict, and the resulting misses are known as conflict misses.

To reduce the likelihood of conflict misses in the cache, it can be designed as a set associative cache. An M KB N-way set associative cache operates in principle as N parallel M/N KB direct mapped caches. When a tag lookup is performed, it is done for each of the N sub-caches. If any sub-cache has a match, there is a cache hit, and data from that sub-cache is returned. If there is a miss, the cache block that was accessed least recently is evicted, and the new data is stored in its place.

Data is never replicated. It is always stored in only one cache block. This allows for the case of having N items that interfere in the cache without incurring any conflict misses. The drawback is that a set-associative cache typically costs more in both hardware and power.

4.3 Copy Back vs. Write Through

When a store instruction writes data to memory, there are two main ways the store can be handled by the cache. In a copy-back cache, the data is written into the cache block but not written to memory, and the cache block is marked dirty. When a dirty cache block is evicted from the cache, its content is different from that of the main memory, and thus must be written back to main memory. If a cache block is not dirty, the main memory content is identical and obviously, does not need to be updated.

In a write-through cache, the data is both written into the cache block as well as main memory. This guarantees that main memory is always coherent with the cache – that is, they always contain identical data. The draw back is the larger number of write transactions that the system has to handle.

4.4 Write-Allocate vs. No Write-Allocate

The previous discussion assumed that when a reference (load or store) misses in the cache, the requested data is fetched and written into the cache, thus evicting another cache block. This kind of cache is known as both read-allocate and write-allocate.

However, some applications and system designs benefit from a no write-allocate cache. In such a cache, a write that misses does not cause data to be fetched into the cache. Instead, the write bypasses the cache and is performed at the next lower level of the memory hierarchy instead. The cache remains unchanged.

In order to prevent the CPU from stalling when such writes miss in the cache, the stores are typically buffered in a write-buffer, that control the writing of the value to memory. This allows the CPU to continue without penalty as long as the write-buffer is not full. Thus, for caches with no write-allocate policies, write-misses often do not cause any performance penalties.

4.5 Cache Misses: The Three Cs

As mentioned in the discussion on set-associativity, cache misses can be divided into three classes according to their causes. These are: compulsory misses, capacity misses, and (as previously mentioned) conflict misses.

- **Conflict misses** are cache misses that occur because more than one piece of data (or code) that is referenced maps to the same cache block, and that miss is not due to the cache being too small. Conflict misses can be eliminated by changing where the data or code is located in memory relative to each other, so that their cache indices are not the same, and thus they do not conflict.

  You will notice conflict misses in the cache analysis display by two or more parallel horizontal bands of red or yellow pixels where two bands occur during the same time period. Using the view→interference grid option, you can see if the two bands indeed reside at the same cache line location.

- **Compulsory misses**, also known as first reference misses, are cache misses that occur because the data, or the data collocated in the same cache block, has not previously been referenced. Thus there is no way the data could have been brought into the cache, and the miss is therefore compulsory.

  You will notice compulsory misses in the cache analysis display by a red or yellow pixel at a specific address the first time that cache block is accessed. Subsequent misses are either capacity or conflict.

- **Capacity misses** are cache misses that occur due to the cache being smaller than the set of data (or code) that is referenced (working set). Capacity misses can be, at least partially, eliminated by restructuring the data to be accessed in smaller chunks, or dividing loops up so that each fits in the cache.

  Capacity misses are normally determined in the cache analysis view by ruling out the first two, conflict and compulsory.

4.6 Techniques to Remedy Cache Misses

There are many techniques for eliminating cache misses, and a large body of literature on each technique. Therefore, only an outline will be discussed in this section.

The first step to remedy cache misses is to identify where the cache misses occur, and the most likely root cause of the cache misses. (Compulsory misses are seldom easy to remedy.)

Conflicts in the Cache

If the cache misses are due to conflicts in the cache, the answer is to move the conflicting pieces of data or code relative to each other (for an example see Step 25 of the Cache Analysis Quick Tour on page 1-17).

- For code this answer is not too complex. By compiling with subsections (‘−mo’) option, each function is put into its own subsection and thus becomes independently relocatable. What remains is to write a suitable link command file that allocates the functions appropriately.
- For data, it is possible to assign data structures to separate sections, but this does require the manual insertion of pragmas.

Capacity Misses

If the cache misses are capacity misses the answer is either to try to access less data or code, or to maximize the reuse of objects. For instance, take the matrix multiply example in Section 5.1. The first version does not reuse the data while it is still in the cache, but goes through the entire array dimension before any reuse is attempted. The second version uses “blocking” or “tiling” to increase the reuse of the array elements before they are evicted from the cache, drastically improving the cache performance.

For code, if a loop is too large to fit in the cache, if possible, it can be split into two loops, one executing after the other. That way, cache misses are only incurred in the first iteration of each of the new loops as opposed to every iteration of the old loop. If it is not possible to split the loop, it might be profitable to trade code size for performance. Given that each instruction cache miss costs at least 5 cycles on the C6211, compiling for smaller code size might actually improve performance, though it is not guaranteed.

Chapter 5

Example Programs

The following example programs are included in the distribution.

Topic **Page**

5.1	Matrix Multiply	5-2
5.2	Conflict Example	5-4
5.3	JPEG	5-5
5.4	Matrix Transpose Operation	5-6
5.5	Vector Operation	5-7

5.1 Matrix Multiply

Located in:

* 6000:

- ❏ C64xx
  <CCS−installation−directory>\examples\sim64xx\cache_analysis\matrix

- ❏ C62xx
  <CCS−installation−directory>\examples\sim62xx\cache_analysis\matrix

This example contains two programs mm1.out and mm2.out both performing matrix multiplication.

mm1.c:

```
short A[128][128], B[128][128], C[128][128];
main()
{
  short i,j,k;
  for (i = 0; i < 128; i++) {
    for (j = 0; j < 128; j++) {
      for (k = 0; k < 128; k++) {
        C[i][j] += A[i][k] * B[k][j];
      }
    }
  }
}
```

mm2.c:

```
short A[128][128], B[128][128], C[128][128];
main()
{
  short i,j,k,ii,jj,kk;
  for (ii = 0; ii < 128; ii+=32) {
    for (jj = 0; jj < 128; jj+=32) {
      for (kk = 0; kk < 128; kk+=32) {
        for (i = ii; i < 32+ii; i++) {
          for (j = jj; j < 32+jj; j++) {
            for (k = kk; k < 32+kk; k++) {
              C[i][j] += A[i][k] * B[k][j];
            }
          }
        }
      }
    }
  }
}
```

Both the programs were built with −o3 −ml −gp options.

The two programs are functionally identical – they both perform a matrix multiply. However, mm2.out is blocked, that is, it works on smaller pieces of the matrices at a time. This reduces the size of the working set and improves temporal locality reducing *capacity misses*. The difference in terms of cache performance is dramatic when the program is simulated and the trace is visualized.

5.2 Conflict Example

Located in:

* 6000:

- ❏ C64xx
  <CCS−installation−directory>\examples\sim64xx\cache_analysis\conflict
- ❏ C62xx
  <CCS−installation−directory>\examples\sim62xx\cache_analysis\conflict

This example consists of two programs with identical source code as seen below. One program (conflict1.out) was linked to illustrate *conflict misses* by linking, the functions avg() and moving() to conflict with each other in the cache. The other program (conflict2.out) was linked to not conflict using a C6416 cache organization.

```
short A[1024], B[1024];
short avg(short *a, short N)
{
  short avg,i;
  for (avg = 0, i = 0; i < N; i++) {
    avg += a[i];
  }
  avg /= N;
  return avg;
}
void moving(short A[], short B[])
{
  int i;
  for (i = 0; i < 1024 – 8; i++) {
    B[i] = avg(&A[i],8);
  }
}
int main()
{
  moving(A,B);
}
```

5.3 JPEG

Located in:

* 6000:

❑ <CCS−installation−directory>\examples\sim64xx\cache_analysis\jpeg

This application is included so as to give a picture of what a real application, as opposed to synthetic examples, may look like when traced and visualized in Cache Analysis. This application is the subject of the Quick Tour earlier in this document. One important feature of this application is that, even though the number of instruction cache misses is relatively low, a large portion of these (using a C6416 cache architecture) are conflict misses.

5.4 Matrix Transpose Operation

Located in:

6000:

- C64xx
  <CCS−installation−directory>\examples\sim64xx\cache_analysis\mat_transpose_oprn
- C62xx
  <CCS−installation−directory>\examples\sim62xx\cache_analysis\mat_transpose_oprn

This example shows how the Cache Analysis Tool can be used to analyze the cache access patterns and then decrease the program cache misses.
For this, it uses two projects; mat_oprn_1.pjt (in .\mat_oprn_1) and mat_oprn_2.pjt(in .\mat_oprn_2).

The example implements the following function on two matrices, A and B, and stores the result in matrix C.

C=A <mat_oprn> Transpose(B)

The size of the matrices are such that the initial version(mat_oprn_1) suffers a lot of data cache misses. The functions mat_xpose_oprn and mat_oprn are placed in such a way that they conflict in the program cache, thus causing a lot of program cache misses.

It is required to analyze the cache access patterns using the Cache Analysis Tool and decrease the total number of cache misses.

The project mat_oprn_2 accomplishes the task by:

- reducing data cache misses by accessing B in parts and reusing them
- reducing program cache misses by placing the conflicting functions together in the memory

5.5 Vector Operation

C5500

0 <CCS-installation

directory>\example\sim55x\cache_analysis\vector_oprn

The example implements the following vector operations in the file “vector_oprn.c”.

- find_mean() − calculate the mean value of a vector
- find_sum_squares() − calculate the sum of squares of values in the vector
- find_dot_product() − calculate the dot products of two vectors

the main() function calls the above three functions in a loop. the functions find_mean(), find_sum_squares() and find_dot_product() are placed in such a way that they conflict in the program cache (refer to lnk1.cmd in the project), thus causing a lot of program cache misses.

The program cache misses are reduced by placing the functions together n the memory (refer to lnk2.cmd in the project directory). To notice the changes, build the project with the linker command file lnk2.cmd.