Simulating RF3 to Leverage Code Tuning Capabilities

Vincent Wan, Pankaj Lal — Texas Instruments, Santa Barbara

ABSTRACT

Using the simulator for software testing and validation helps reduce time-to-market. Porting an application to the simulator is a straightforward exercise. This application note provides an example using the Reference Frameworks Level 3 application and the 'C6416 device cycle accurate simulator.

In addition to testing the application prior to the release of the production silicon, the simulator provides visibility into the real-time behavior of an application through basic code tuning capabilities provided in Code Composer Studio 3.0 and the Analysis ToolKit (ATK). In particular, this application note looks at the RF3 application's cache usage and profiles the CPU cycles in the application's functions. Ways to optimize the application are suggested.

Contents

1 Introduction 2
2 Porting RF3 to the DSK6416 Simulator 2
2.1 Requirements 3
2.2 Changing the Driver 3
2.3 Application Modifications 4
2.4 Building the Application 6
2.5 Using Port and Pin Connect to Run the Application 6
2.6 For Your Own Application 8
3 Basic Profiling Using Tuning Tools and the Analysis ToolKit 9
3.1 Examining Cache Usage 9
3.2 Obtaining the Distribution of CPU Cycles Spent in RF3 Using the ATK 11
3.3 Optimization Example 13
4 Conclusion 15
5 References 15
Appendix A: List of Accompanying Files 16
Appendix B: Considerations When Porting to Other Devices 16
Appendix C: Caveats 17

Figures

Figure 1. IOM Codec Driver Layout 3
Figure 2. Setup of Port and Pin Connect 6
Figure 3. Timing Diagram for the Start of an Audio Frame 7
Figure 4. Profile Setup for Cache Usage Profiling 9
Figure 5. CacheTune Graph Showing Cache Access Pattern 10

Tables

Table 1. Frame Sync Settings for Selected Audio Frequencies 7
Table 2. Distribution of CPU Cycles While Processing One Frame 12
Table 3. Effects of Optimization 15
Table 4. Frame Processing Time Improvement 15

1 Introduction

Reducing time-to-market is an important factor in determining the success or failure of a product. You can move toward this goal by minimizing the development time after silicon is made available. Reusing DSP components, such as those provided in Reference Frameworks Level 3 (RF3), helps to reduce the time and effort spent to release a new DSP software application. Moreover, you can validate your software on a simulator, so that most of the development work can be done prior to the silicon release.

This application note describes how to run and validate an RF3-based application on the 'C6416 device cycle accurate simulator. We'll show how to obtain better visibility into the behavior of an application using basic code tuning capabilities provided in Code Composer Studio (CCStudio) 3.0 and the Analysis ToolKit (ATK). In particular, we'll look at the application's cache usage and profiling of CPU cycles in the application's functions.

By following a similar procedure for your applications, you can write your system software early, so that when a DSK or EVM becomes available, the full application can run on the hardware with minimal changes.

Some familiarity with DSP/BIOS and RF3 is recommended. For details, please see *Reference Frameworks for eXpressDSP Software: RF3, A Flexible, Multi-Channel, Multi-Algorithm, Static System* (Reference 1). You can also learn more about application code tuning (ACT) by referring to the CCStudio online help.

2 Porting RF3 to the DSK6416 Simulator

In this section we describe how to modify an existing RF3 application so that it works on the 'C6416 device cycle accurate simulator. The end-goal is to have a complete RF3 application with EDMA, MCBSP, and TIMER peripherals all being exercised in the same fashion as on the DSK6416, yet with very minimal code changes.

We have chosen to use the device cycle accurate simulator for this application note, because it is capable of reporting CPU cycle-based information for the CPU and the device, by including all stalls, memory latency, and system effects. However, if you are only interested in running the application or in using CacheTune to look at cache usage behavior, you can use the device functional simulator instead to speed up your simulation.

2.1 Requirements

The procedures in the following sections assume you have installed the following software:

- **Code Composer Studio 3.0** (http://focus.ti.com/docs/toolsw/folders/print/ccstudio.html) This includes DSP/BIOS 4.90. If you have installed a more recent version of DSP/BIOS, see the GettingStartedGuide.html file for changes you will need to make to projects.
- **CCStudio v3.00.01.00 patch** (https://www-a.ti.com/downloads/sds_support/CCS_3.0.1-UA.htm)
- **Reference Frameworks 2.2 or greater**—installed in a location to be referred to as *RF_DIR* (https://www-a.ti.com/downloads/sds_support/targetcontent/RF/index.html)
- **Device Driver Kit (DDK) 1.1**—installed in a location to be referred to as *DDK_DIR* (https://www-a.ti.com/downloads/sds_support/targetcontent/DDK/index.html)
- **Analysis ToolKit 1.31** (https://www-a.ti.com/downloads/sds_support/C6000-3.0-SA-to-Analysis Tool Kit 1.31.0.htm)
- **Microsoft Office including Microsoft Excel**

2.2 Changing the Driver

As shown in Figure 1, RF3 uses the IOM driver module dsk6416_edma_aic23 to configure the hardware TLV320AIC23 audio codec. Since this audio codec does not exist in the simulator (it is an on-board peripheral on the 'C6416 DSK), we need to remove the setup for this codec.

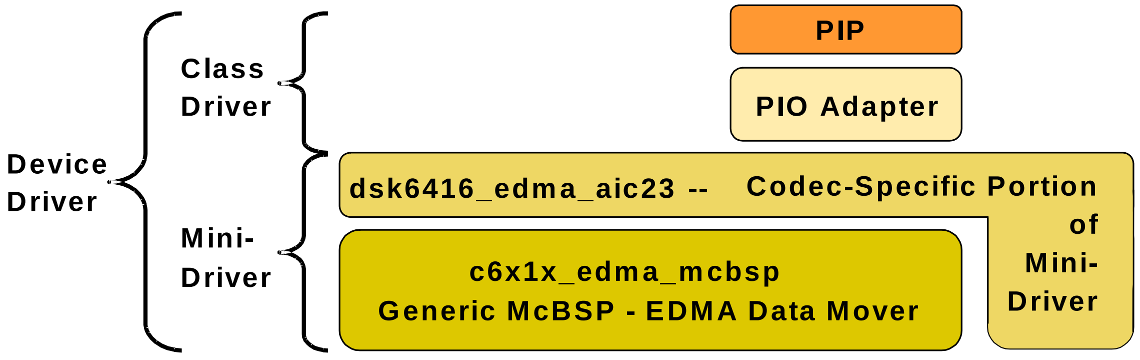

PIP
PIO Adapter
Class Driver
Device Driver
Mini-Driver
dsk6416_edma_aic23 -- Codec-Specific Portion of Mini-Driver
c6x1x_edma_mcbsp
Generic McBSP - EDMA Data Mover

Figure 1. IOM Codec Driver Layout

To remove the codec setup, follow these steps:

1. Copy the project file *DDK_DIR*\ddk\src\audio\dsk6416\dsk6416_edma_aic23.pjt to *DDK_DIR*\ddk\src\audio\dsk6416\dsk6416_edma_**null**.pjt.
2. Import the 'C6416 Device Cycle Accurate Simulator (Little Endian) into CCSetup using the Import Configuration dialog. See the CCStudio Setup online help if you need further instructions on how to do this.
3. Open the new project in CCStudio.

4. Comment out lines 163 and 164 in dsk6416_edma_aic23.c:

```
/* Set codec parameters (this will also initialize the codec) */
//    if (!AIC23_setParams(&(params->aic23)))
//        return IOM_EALLOC;
```

5. Change the build options for the Archiver so that the Output Filename is ..\..\..\lib\dsk6416_edma_**null**.l64.
6. Do a Rebuild All. This builds a new library file *DDK_DIR*\ddk\lib\dsk6416_edma_null.l64.
7. Copy the new library file to the *RF_DIR*\referenceframeworks\lib directory.

2.3 Application Modifications

To modify the application, we first need to change the RTDX protocol to use simulator mode instead of JTAG. Any application needs to make this change in order to be run on the simulator because some RTDX operation mechanisms change when using the simulator. This has no significant impact on the application's general behavior.

1. On line 38 of *RF_DIR*\referenceframeworks\apps\rf3\dsk6416\appBoard.tci, add the line shown in bold below:

```
if (APPRTDXAVAILABLE) {
    bios.enableRtdx(prog);
    tibios.RTDX.MODE = "Simulator";
}
```

2. Set the CPU frequency used by DSP/BIOS statistics objects by adding the following line in the same appBoard.tci file. This application note assumes we are simulating the behavior on a 600 MHz 'C6416 DSK.

```
tibios.GBL.CLKOUT = 600;
```

3. Rebuild the DSP/BIOS configuration by executing the following command from a DOS box. (Replace "CCStudio" with your CCStudio 3.0 installation path if different.)

```
> cd RF_DIR\referenceframeworks\apps\rf3\dsk6416
> c:\CCStudio\bin\utilities\tconf\tconf -Dconfig.tiRoot="C:\CCStudio"
 -Dconfig.platform="Dsk6416" -Dconfig.importPath="../appConfig;../../../include/tci"
 appcfg.tcf
```

When porting your own application to your simulator, the following application simplifications can be omitted. However, to simplify our discussion in this application note, we have made the following changes to the base RF3 application:

- Move all code and data into internal memory. This allows us to concentrate on a smaller memory region when using CacheTune, making the activity easier to view without requiring you to continuously scroll up and down, because external memory is located at addresses above 0x80000000.
- Use allpass filter coefficients in RF3 instead of the default lowpass and highpass filter coefficients. This makes it easier to validate the output data against the input data. The output samples should be identical to the input samples except for their least significant bit, which can still vary due to truncation errors.

You can follow this procedure to make these simplifications to the RF3 application:

1. In CCStudio, open *RF_DIR*\referenceframeworks\apps\rf3\dsk6416\app.pjt.
2. In *RF_DIR*\referenceframeworks\apps\rf3\appModules\appThreads.c, change the call to ALGRF_setup to the following. This ensures that only the internal heap allocated in ISRAM is used for XDAIS algorithm buffers.

```
ALGRF_setup( INTERNALHEAP, INTERNALHEAP );
```

3. In the same file, comment out the line that shows external heap usage:

```
// UTL_showHeapUsage( EXTERNALHEAP );
```

4. In *RF_DIR*\referenceframeworks\apps\rf3\appModules\thrAudioproc.c, comment out the existing filter coefficients and replace them with allpass coefficients as follows:

```
static Sample filterCoefficients[ NUMCHANNELS ][ 32 ] = {
    /*
     * The following filters have been designed with the compromise to let
     * through a reasonable amount of audio content when used with sampling
     * rates from 8 to 48 kHz. This is due to the wide variety of default
     * sampling rates supported in the various ports of RF5.
     */

    /* Low-pass, 32 taps, passband 0 to 500 Hz for sampling rate of 8kHz */
/*    {
        0x08FC, 0xF6DE, 0xF92A, 0xFA50, 0xFB17, 0xFBF0, 0xFD2A, 0xFECF,
        0x00EC, 0x036C, 0x0623, 0x08E1, 0x0B6E, 0x0D91, 0x0F1A, 0x0FE9,
        0x0FE9, 0x0F1A, 0x0D91, 0x0B6E, 0x08E1, 0x0623, 0x036C, 0x00EC,
        0xFECF, 0xFD2A, 0xFBF0, 0xFB17, 0xFA50, 0xF92A, 0xF6DE, 0x08FC
    }, */
    /* High-pass, 32 taps, passband 500 Hz to 4 kHz for sampling rate of 8kHz */
/*    {
        0x08B6, 0xFC32, 0xFC41, 0x09CF, 0x0467, 0x0B2E, 0x0099, 0x05FB,
        0xF920, 0x014E, 0xF1B4, 0xFF3F, 0xE8F9, 0x02AB, 0xD626, 0x41EB,
        0x41EB, 0xD626, 0x02AB, 0xE8F9, 0xFF3F, 0xF1B4, 0x014E, 0xF920,
        0x05FB, 0x0099, 0x0B2E, 0x0467, 0x09CF, 0xFC41, 0xFC32, 0x08B6
    },*/
    {
        // Allpass, 32 taps
        0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
        0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
        0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
        0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x7FFF
    },
    {
        // Allpass, 32 taps
        0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
        0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
        0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
        0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x7FFF
    },
};
```

2.4 Building the Application

Follow these steps to build the application:

1. Make sure the *RF_DIR*\referenceframeworks\apps\rf3\dsk6416\app.pjt project is open in CCStudio.
2. Open the link.cmd file. Change the following line:

```
-l dsk6416_edma_aic23.l64
```

to:

```
-l dsk6416_edma_null.l64
```

3. Rebuild the project. Make sure there are no errors.

2.5 Using Port and Pin Connect to Run the Application

Simulators do not support the simulation of audio codecs that are external to the 'C6416 device—like the AIC23 codec. This limitation can be overcome using the Pin Connect and Port Connect plugins in CCStudio 3.0. They help set up the control and data interaction between the serial port and the codec.

The download available with this application note provides a GEL file to set up Port Connect to supply data to the Data Receive register (DRR) register of the McBSP and to accept data from the Data Transmit Register (DXR) register. The GEL file also uses Pin Connect to supply the receive and transmit clock (CLKR, CLKX) signals and the receive and transmit frame syncs (FSR, FSX). Figure 2 shows the setup.

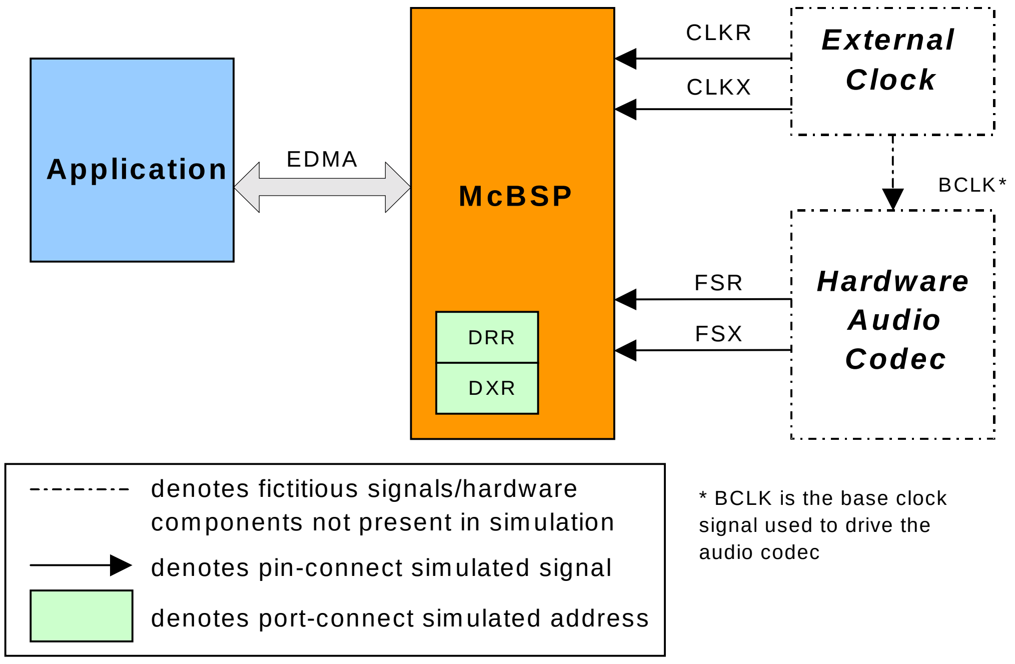

Application
EDMA
McBSP
DRR
DXR
CLKR
CLKX
External Clock
BCLK*
FSR
FSX
Hardware Audio Codec
denotes fictitious signals/hardware components not present in simulation
denotes pin-connect simulated signal
denotes port-connect simulated address
* BCLK is the base clock signal used to drive the audio codec

Figure 2. Setup of Port and Pin Connect

Note that in the original RF3 application two McBSPs are used: one for control/setup of the audio codec, and the other for data streaming. In the simulated application we only need one McBSP for data streaming, as we have modified the driver to omit the setup of the codec.

The GEL file (appSimulation.gel) sets up both Port and Pin Connect with the correct signals to drive the McBSP with a 12 MHz external clock, which is the same as the one available on the 600 MHz DSK6416), and frame syncs corresponding to a 44.1 kHz sampling rate.

If desired, you can change the sampling frequency by adjusting the frame sync periods in the FSR2.txt and FSX2.txt files. For example, if you have a 600 MHz DSK6416 and wish to run your GSM application (sampling rate of 16 kHz) on the simulator, you would calculate the frame sync period as follows:

```
Frame sync period (in cycles) = 600,000,000 Hz / 16,000 Hz
```

During this period, the signal needs to stay high for one cycle of the external clock and then go low for the remainder of the period. Figure 3 shows how each frame sync marks the beginning of an audio frame on the AIC23 audio codec, when it is configured in DSP audio-interface mode. Refer to the documentation on the audio codec [Reference 2] if you want to learn more about how the codec synchronizes with the McBSP. Note that when you specify the frame sync signal in Pin Connect, the units are in # CPU cycles, hence every cycle of the external clock would correspond to 50 CPU cycles (600 MHz/ 12 MHz = 50).

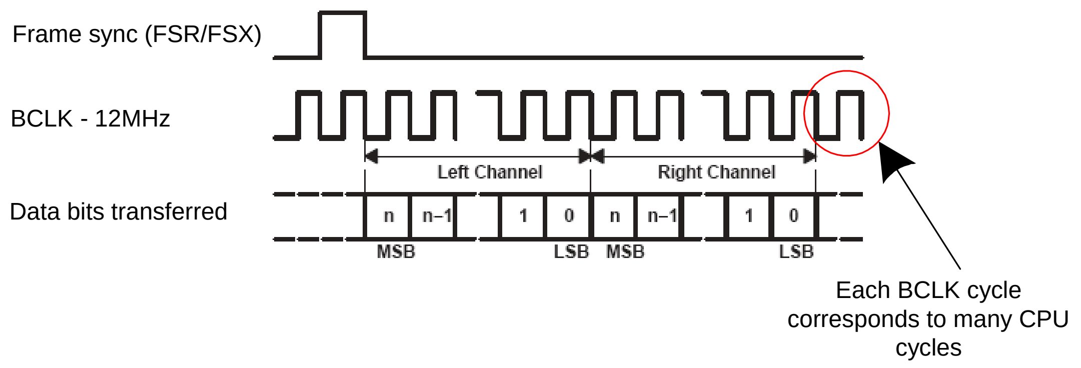


Figure 3. Timing Diagram for the Start of an Audio Frame

The frame sync settings for use in the FSR2.txt and FSX2.txt files are shown in Table 1 for some standard audio frequencies:

Table 1. Frame Sync Settings for Selected Audio Frequencies

Sampling Frequency (kHz)	Frame Sync Period (in # CPU Cycles)	Pin Connect Settings (Parameters in # CPU Cycles)
8	75000	([+50,1] [+74950,0]) rpt EOS*
16	37500	([+50,1] [+37450,0]) rpt EOS
44.1 (default)	13605	([+50,1] [+13555,0]) rpt EOS
48	12500	([+50,1] [+12450,0]) rpt EOS
96	6250	([+50,1] [+6200,0]) rpt EOS

* "rpt EOS" means “Repeat until End Of Simulation”, meaning the signals will toggle indefinitely.

To run the application, follow these steps:

1. Open the appSimulation.gel file supplied in the download available with this application note. Modify all paths in the Connect() hotmenu item to point to the location that contains the .txt files listed in Appendix A, which are also supplied in the download for this application note.
2. Use **File->Load GEL** to load the GEL file.
3. Reset the DSP using **Debug->Reset CPU**.
4. Load the executable app.out.
5. Select **GEL->Application Control->Connect** to connect all pins and ports. You should see a message in the output window indicating the success of the operation. Note that there is also a GEL option for disconnecting all pins and ports.
6. Run your executable (app.out) by pressing F5.
7. You can open the Message Log and the Statistics View to make sure the application is running correctly at the desired data rate. Right-click on the Statistics View and choose Property Page. In the Units tab, choose milliseconds for the stsTime objects.

   The default sampling rate of 44.1 kHz corresponds to a frame rate of 1.81 ms which should be reported by stsTime0 if you are simulating a 'C6416 running at 600 MHz.
8. You can also verify the output of the application by opening the output_data.txt file in a text editor. The pattern will start with a sequence of zeros (silence) due to the priming buffers. Then it will cycle through the data pattern defined in input_data.txt. The last digit might not match because of truncation errors introduced during the application of the FIR filter.

This completes the RF3 modifications needed to run on the simulator. As you can see, only minor driver and application modifications were required. A new GEL file was also introduced as the Port and Pin Connect proxy for the hardware codec.

2.6 For Your Own Application

Even if your application is not based on RF3, the key lesson here is that you will need to make changes to your driver to remove or disable code that configures and drives off-chip peripherals and peripherals not supported by the simulator. Then, by using Pin Connect to simulate external signals and Port Connect to connect a memory (port) address to a file, you can perform a full system simulation of your application.

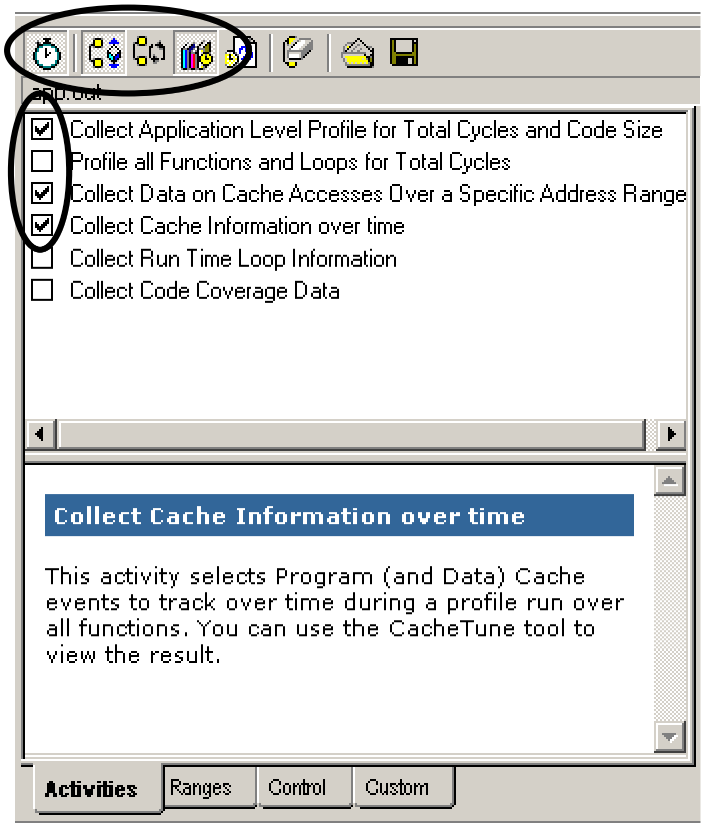
3 Basic Profiling Using Tuning Tools and the Analysis ToolKit

The great thing about having an application run on the simulator is that we can gather information about the application that we cannot easily obtain by running the application on actual hardware. This information includes, for example, a very detailed cycle count breakdown, cache usage information, and memory access behavior information.

3.1 Examining Cache Usage

Let’s take a closer look at how the cache is used in this application. To do this, use the following procedure. Statistics given are for the RF3 application modified to run with the simulator as described in this application note.

1. Run the application as described on page 8 up to the point where you have connected the pins and ports, but have not yet run the application.
2. Select **Profile->Setup**. Set up profiling by pressing the three tool icons and checking the three checkboxes in the Activities tab as shown in Figure 4. This enables profiling on all functions and collection of cache and cycle information.



[✓] Collect Application Level Profile for Total Cycles and Code Size
[ ] Profile all Functions and Loops for Total Cycles
[✓] Collect Data on Cache Accesses Over a Specific Address Range
[✓] Collect Cache Information over time
[ ] Collect Run Time Loop Information
[ ] Collect Code Coverage Data

Collect Cache Information over time

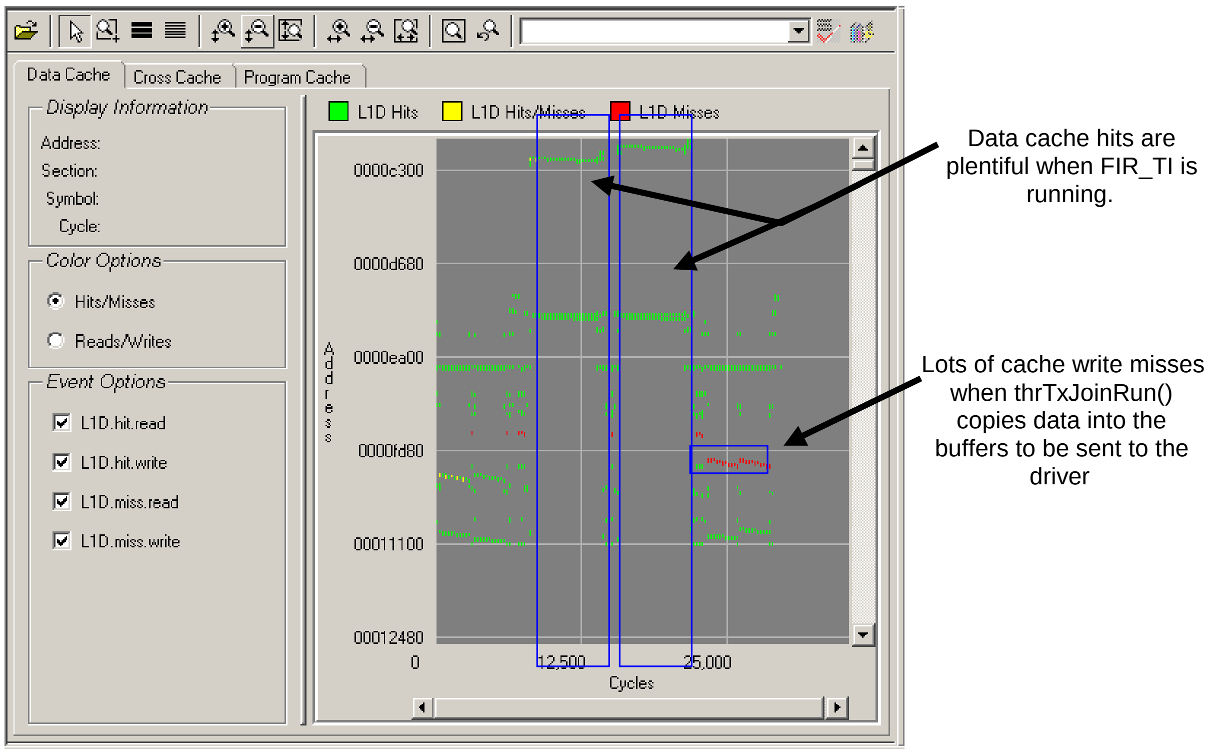
This activity selects Program (and Data) Cache events to track over time during a profile run over all functions. You can use the CacheTune tool to view the result.

Figure 4. Profile Setup for Cache Usage Profiling

3. Select **Profile->Viewer** and **Profile->Tuning->CacheTune** from the menus.
4. Set a breakpoint on the first line of the function thrRxSplitRun in file thrRxSplit.c.
5. Run the code to this breakpoint. In this procedure we want to skip over the one-time initialization code and profile only the code that processes audio in "steady-state".

6. Run the code once more until it reaches this breakpoint again. This runs through the processing code once and populates the cache so that we eliminate all compulsory misses that happen only when processing the first frame.
7. Disable profiling in the Profile Setup window by clicking the clock icon. Then re-enable it. (This is a workaround for SDSsq39604 to reset the Profiler so that it does not profile what happens before the breakpoint).
8. Set a breakpoint on the last line of thrTxJoinRun() in thrTxJoin.c.
9. Run to this breakpoint.
10. Disable profiling. The CacheTune graph is updated at this point.
11. Click the icon repeatedly (about 11 times) to zoom in on the lowest numbered addresses along the top of the CacheTune graph in the Data Cache tab. Figure 5 shows the cache access pattern for processing an audio frame. You can see the cache hit and miss patterns corresponding to memory accesses performed during processing. You can ignore misses at addresses greater than 01b40000; those are register addresses that are not cached, so every access shows up as a cache miss.

    Overall, read cache misses should be few and far between. In particular, the data accesses performed by the FIR_TI algorithm are well-cached. This is because we are looking at the access pattern during processing of the second frame, and the cache line never had to be evicted between frames. However, we are not so lucky with the cache write misses that occur while audio data is interleaved in thrTxJoinRun(). Yet, we are generally less worried about write misses on the 'C64x platform because they do not automatically translate into memory stalls.


Data Cache | Cross Cache | Program Cache

Display Information
Address:
Section:
Symbol:
Cycle:

Color Options
[✓ Hits/Misses]
Reads/Writes

Event Options
☑ L1D.hit.read
☑ L1D.hit.write
☑ L1D.miss.read
☑ L1D.miss.write

🟩 L1D Hits
🟨 L1D Hits/Misses
🟥 L1D Misses
Address
0000c300
0000d680
0000ea00
0000fd80
00011100
00012480
0
12,500
25,000
Cycles
Data cache hits are plentiful when FIR_TI is running.
Lots of cache write misses when thrTxJoinRun() copies data into the buffers to be sent to the driver

Figure 5. CacheTune Graph Showing Cache Access Pattern

12. Move to the program cache tab to see the L1P cache misses. Again cache usage seems well-optimized overall.
13. If you'd like to quantify the cache misses, look at the Summary pane in the profile viewer (**Profile->Viewer**). The percentages of L1P and L1D stall cycles with respect to the total number of cycles simulated are below 2%. You can also see many other useful statistics, such as the # of NOP cycles, # of branches taken, # of stores and loads. Note that only the Summary pane is reliable in a multi-threaded DSP/BIOS application, as the function profiler currently works only for "straight-line" code such as a simple MP3 algorithm test harness (SDSsq39341).

In summary, it appears the placement of data buffers in RF3 does not cause a significant amount of unnecessary cache misses, although we didn't do anything special. In a way, this is expected because RF3 has a relatively low footprint and its data buffers of 320 bytes are small relative to the L1D cache size of 16 KB.

Guidelines to improve the cache hit percentage are provided in the CCStudio Tuning Tools online help, and include:

- Align buffers on 64-byte boundaries, for example, by using the DATA_ALIGN pragma. The 64-byte size is the 'C64x L1D cache line width.
- Chop large buffers into pieces that are smaller than 16 KB so that the L1D cache is not continually thrashed.

Refer to the *TMS320C6000 DSP Cache User's Guide* (Reference 3) if you are interested in learning more about the cache and how to better optimize it.

3.2 Obtaining the Distribution of CPU Cycles Spent in RF3 Using the ATK

Another way to get useful statistics is through the Analysis ToolKit (ATK). This kit can provide code coverage information as well as the number of exclusive cycles spent on each function and source line. Exclusive cycle count is used to designate the number of cycles spent in a function or on a source line while excluding any cycles spent inside called sub-functions. When an interrupt happens, the interrupt service routine (ISR) is treated as a function call and events that happen are counted towards the ISR itself. Moreover, the ATK can also give exclusive counts of events such as cache misses, cross-path stalls, and external memory accesses.

Note that we use the ATK instead of the CCStudio Profiler because we are profiling a threaded (DSP/BIOS) application. The ATK logs a count of PC addresses; the CCStudio Profiler tracks function entry and exit points. The CCStudio Profiler is not suited to handling asynchronous interrupts and pre-emption since its exclusive counts incorrectly include all pre-emption activity.

By default, the ATK displays exclusive counts for 12 different types of events. We focus only on the number of cycles spent executing application, framework, and driver functions.

First we profile the cycle distribution across all functions during initialization using the following procedure:

1. Select **Profile->Setup**. Check the "Collect Code Coverage Data" box in the Activities tab.
2. Select **Debug->Reset CPU**.
3. Load the program and enable the profiler in the Profile Setup.

4. Load the appSimulation.gel file.
5. Select **GEL->Application Control->Connect** to connect all pins and ports. You should see a message in the output window indicating the success of this operation.
6. Set a breakpoint on the first line of the thrRxSplitRun function in file thrRxSplit.c.
7. Run the code to this breakpoint. This means the application has run through all of the initialization code and the driver has just returned the first data buffer.
8. Disable profiling in the Profile Setup.
9. Select **Profile->Coverage Viewer**. This launches a Microsoft Excel spreadsheet that shows the coverage and profile information.

The profile data in the spreadsheet corresponds to the events that occurred during the initialization stage. Of course, we would prefer to see the execution cycles for processing one audio buffer during frame-based processing. To do this, use the following procedure:

1. Close the Excel spreadsheet.
2. Select **Debug->Reset CPU**.
3. Reload the program. Enable the profiler in Profile Setup.
4. Set a breakpoint on the first line of the thrRxSplitRun function in file thrRxSplit.c.
5. Run the code to this breakpoint. This means the application has run through all of the initialization code and the driver has just returned the first data buffer. Run it once more as we want to profile the processing of the second frame, when the application is in "steady-state" and processing code and data has been cached.
6. Disable profiling, then re-enable it to flush out profile information obtained during initialization and processing of the first frame.
7. Run the code until it hits the same breakpoint. At this point, you have profiled the period from when the second frame becomes ready to when the third frame becomes ready.
8. Disable profiling.
9. Select **Profile->Coverage Viewer**. This launches an Excel spreadsheet that shows the coverage and profile information.

Table 2 shows the breakdown of function cycle counts when RF3 is processing a frame. Note that it also includes activity that occurs while the application is waiting for a second frame to become ready.

Table 2. Distribution of CPU Cycles While Processing One Frame

	Function	File	# Times Called	Cycle.Total
Algorithm-execution-related code	ALGRF_activate	algrf_activate.c	4	124
	ALGRF_deactivate	algrf_deactivate.c	4	124
	FIR_TI_activate	fir_ti_ialg.c	2	40
	FIR_TI_deactivate	fir_ti_ialg.c	2	40
	FIR_TI_filter	fir_ti_filter.c	2	123
	FIR_TI_gen	fir_ti_filter.c	2	11,096
	FIR_apply	fir.c	2	121

	Function	File	# Times Called	Cycle.Total
	VOL_TI_amplify	vol_ti_ivol.c	2	95
	scale	vol_ti_ivol.c	2	418
	VOL_apply	vol.c	2	114
Driver-related code	transfer	pio.c	6	387
	PIO_rxPrime	pio.c	2	90
	PIO_txPrime	pio.c	2	122
	rxCallback	pio.c	1	82
	txCallback	pio.c	1	62
	mdSubmitChan	c6x1x_edma_mcbsp.c	2	328
	linkPacket	c6x1x_edma_mcbsp.c	2	622
	isrCommon	c6x1x_edma_mcbsp.c	2	180
	isrOutput	c6x1x_edma_mcbsp.c	1	174
	isrInput	c6x1x_edma_mcbsp.c	1	150
Thread code	thrAudioprocRun	thrAudioproc.c	2	515
	thrControlIsr	thrControl.c	1	86
	thrRxSplitRun	thrRxSplit.c	1	5286
	thrTxJoinRun	thrTxJoin.c	1	5096
UTL-related code	UTL_stsPeriodFunc	utl_stsPeriod.c	2	125
	UTL_stsPhaseFunc	utl_stsPhase.c	1	45
	Others (including idle time)		**1,080,335**	
	Total (# cycles elapsed between 2 frames)		**1,105,980**	

The ATK profiler shows us that most of the cycles are spent in the FIR_TI_gen() function, where FIR filtering is performed. Lots of cycles are included as part of the "Others" category, as this encompasses all interrupt service routines and DSP/BIOS-related functions, including the idle loop in which application idle time is spent.

3.3 Optimization Example

Now that you have seen the power of the tools that deliver valuable statistics about the application, you can use this information to optimize your own code. Typical optimization techniques include:

- Adjusting buffer placement and alignment to improve cache usage
- Reducing the number of instructions in critical functions where the application spends the most cycles
- Turning on compiler optimization flags to improve performance

It is beyond the scope of this application note to cover these techniques in detail; however, we can give you a better appreciation of the value of these simulation tools by showing the results from the following optimizations:

- Enable the –o3 compiler optimization flag.
- Remove debug information from the FIR_TI algorithm by removing the –g compiler flag.

- Rewrite the FIR_TI_gen() function in *RF_DIR*\referenceframeworks\src\fir_ti\fir_ti_filter.c by adding the following code shown in bold:

```
void FIR_TI_gen(XDAS_Int16 *in, XDAS_Int16 *coeff, XDAS_Int16 * restrict out,
XDAS_Int16 nCoeff, XDAS_Int16 nout)
{
    XDAS_Int16  i, j;
    XDAS_Int32  sum;

    _nassert((unsigned)out   % 8 == 0);
    _nassert((unsigned)in    % 8 == 0);
    _nassert((unsigned)coeff % 8 == 0);

    #pragma MUST_ITERATE(,,4);
    for (j = 0; j < nout; j++)
    {
        sum = 0;

        #pragma MUST_ITERATE(4,,4);
        for (i = 0; i < nCoeff; i++)
            sum += (XDAS_Int32)(in[i + j]) * (XDAS_Int32)(coeff[i]);

        out[j] = sum >> 15;
    }
}
```

  – **The restrict keyword** helps the compiler determine memory dependencies. Its use on an output function parameter represents a guarantee by the programmer that, within the scope of the pointer declaration, the object pointed to can be accessed only by that pointer. This practice helps the compiler optimize certain sections of code because aliasing information can be more easily determined.
  – **The MUST_ITERATE and _nassert statements** tell the compiler that the inner loop executes at least 4 times and that the outer and inner loops run a multiple of 4 times. This causes the compiler to produce code that operates on multiple data values with a single instruction, also known as SIMD (Single Instruction Multiple Data) optimization.

The compiler can use all this information to generate the best loop possible by unrolling better, while using the LDDW instruction on the 'C64x architecture to load four 16-bit values at a time from the coefficient and input data arrays.

The downside of this optimization is that there is an extra restriction that the coefficient, input, and output data arrays must be 8-byte aligned. This is automatically satisfied by any statically allocated array, and requires requesting an alignment of 8 for the working buffer workbuf in FIR_TI_alloc() of *RF_DIR*\referenceframeworks\src\fir_ti\fir_ti_ialg.c:

```
    memTab[WORKBUF].alignment   = 8;
```

Furthermore, another restriction is that the coefficient and output data arrays must now be of sizes that are multiples of 4 for the unrolling to be performed.

After rebuilding FIR_TI with the new options, and linking the application with the new module, we obtained the results in the right column of Table 3 after profiling the second frame's processing, from the first line of thrRxSplitRun() to the last line of thrTxJoinRun():

Table 3. Effects of Optimization

Function	File	#times called	cycle.Total Before Optimization	cycle.Total After Optimization
FIR_TI_activate	fir_ti_ialg.c	2	40	16
FIR_TI_deactivate	fir_ti_ialg.c	2	40	16
FIR_TI_filter	fir_ti_filter.c	2	123	70
FIR_TI_gen	fir_ti_filter.c	2	11,096	2,603
Total			**11,299**	**2,705**

Hence, this optimization has resulted in a 76% improvement in the total number of cycles spent executing the FIR_TI algorithm on each frame. But one would ask: how does this affect the overall picture? Table 4 shows the improvement over the processing time for an entire frame.

Table 4. Frame Processing Time Improvement

	Processing Time for One Frame (# CPU Cycles)
Before optimization	29,605
After optimization	20,996
% improvement	29.1%

So indeed, we have improved the overall application efficiency significantly by focusing on optimizing the FIR_TI algorithm.

For more information on using the compiler to better optimize your code, refer to the *TMS320C6000 Optimizing Compiler User's Guide* [Reference 4].

4 Conclusion

Porting an existing RF3-based application to the simulator is a straightforward exercise that gives enhanced visibility into the behavior of the application, including but not limited to cycle count distribution across the application, cache usage, memory accesses, and related stalls. Armed with this information, you can better optimize your code. Furthermore, you can develop your product before silicon and development boards become available, and can thus deliver your product to the market earlier.

5 References

1. *Reference Frameworks for eXpressDSP Software: RF3, A Flexible, Multi-Channel, Multi-Algorithm, Static System* (SPRA793, Texas Instruments)
2. *TLV320AIC23B Data Manual* (SLWS106H, Texas Instruments)
3. *TMS320C6000 DSP Cache User's Guide* (SPRU656, Texas Instruments)
4. *TMS320C6000 Optimizing Compiler User's Guide* (SPRU187, Texas Instruments)

Appendix A: List of Accompanying Files

The following list describes the files in the zip file that accompanies this application note. See Section 2.5, "Using Port and Pin Connect to Run the Application" for more about these files.

- **CLKR2.txt.** Pin Connect file to specify the external clock period on the receive side. A 12 MHz clock is assumed. That is, the period is 600 MHz/12 MHz = 50 CPU cycles per external clock cycle.
- **CLKX2.txt.** Pin Connect file to specify external clock period on the transmit side. A 12 MHz clock is assumed. That is, the period is 600 MHz/12 MHz = 50 CPU cycles per external clock cycle.
- **FSR2.txt.** Pin Connect file to specify the frame sync period on the receive side. Assumes a sampling rate of 44.1 kHz.
- **FSX2.txt.** Pin Connect file to specify the frame sync period on the transmit side. Assumes a sampling rate of 44.1 kHz.
- **input_data.txt.** Contains an input data sequence to be read from the DRR, used by Port Connect. It is repeated indefinitely.
- **output_data.txt.** Contains an output data sequence that has been written to the DXR during execution. Used by Port Connect.
- **appSimulation.gel.** GEL file used to facilitate the setup of pin and port connects. This file adds CCStudio menu options to make it easier to connect and disconnect the various pins and ports necessary for data streaming to occur. It needs to be customized to point to the location of the .txt files.

Appendix B: Considerations When Porting to Other Devices

If, for example, you want to port this application to a 'C6713, you need to consider the following issues:

- **Modify the addresses of registers** in appSimulation.gel in GEL_PortConnect(). The 0x38000000 corresponds to DRR2 on the 'C6416. For example, if you are using DRR1, the address would be 0x34000000 as noted in the 'C6713 datasheet.
- **Modify GEL_PinConnect() statements** in appSimulation.gel. For example, if you are using MCBSP 0 or 1 instead of MCBSP 2 for data streaming, you would make the corresponding change in appSimulation.gel.
- **Optimize differently.** For example, you would chop data buffers into sizes of less than 4 KB if possible, since 'C6x1x devices have only 4 KB of L1D cache. The 'C64x devices, in comparison, have 16 KB of L1D cache.
- **Modify timing.** The external clock period and the frame sync period are set in the pin connect files; the settings depend on your device's clock frequency. See Section 2.5 and Appendix A to find out how these quantities are computed.

Appendix C: Caveats

Timing on the device cycle accurate simulator is only 90-95% accurate, relative to real hardware.

We have noticed that during the initialization of RF3, the EDMA is slightly slower on the simulator than on hardware (relative to the CPU clock speed). This has caused the small dummy "loop" job relied upon by the inner workings of the c6x1x_edma_mcbsp driver to finish only after the first two EDMA transfers are submitted on the input side. This triggers an undefined behavior in the driver, which causes the first two EDMA jobs to be re-submitted. Hence, if you profile the initialization code with the simulator, notice it takes much longer for the code to reach the first call to thrRxSplitRun() than on actual hardware.

One way to avoid this is to wait for the "loop" job to finish before submitting the second EDMA job. You can do this by adding a delay to the PIO_rxStart() function in pio.c of the DDK as follows:

```
/*
 * ======== delay ========
 */
Void delay( unsigned long t )
{
    volatile unsigned long i;
    for( i=0; i<t; i++ ) {
       asm ( " NOP ");
    }
}

/*
 *  ======== PIO_rxStart ========
 *  submit frames to receiver
 *
 *  CONSTRAINTS:
 *  -  PIO_rxStart() may only be called in main() before interrupts
 *     are enabled
 */
Void PIO_rxStart (PIO_Handle pio, Uns frameCount)
{
    for ( ; frameCount > 0; frameCount--) {
        delay(10);  // Delay value needs to be adjusted depending on CPU speed
        transfer(pio, PIP_getWriterNumFrames(pio->pip));
    }
}
```

Note that the delay shown assumes a 600 MHz 'C6416. For other processor speeds, you might need to vary the delay value to obtain a small delay that is just large enough that the "loop" job has time to complete its transfer before the second EDMA job is submitted through another call to transfer().