Texas Instruments

Application Report
SPRA924A − April 2005

Achieving Efficient Memory System Performance with the I-Cache on the TMS320VC5501/02

Cesar Iovescu, Cheng Peng Software Applications
Miguel Alanis, Gustavo Martinez Hardware Applications

ABSTRACT

The TMS320VC5501 and TMS320VC5502 were designed as low cost DSPs for low cost real-time solutions. In order to achieve the best performance with these processors the system designer must follow the guidelines provided in the application report.

1. Execute code from external memory with the on-chip I-Cache enabled.
2. Locate data in internal memory at run time.
3. Use the External Memory Interface (EMIF) at its maximum clock frequency

The application report will present the performance of several popular algorithms implemented on the C5502: G.729AB and G.723.1 vocoders for telecommunications, MP3 decoder for audio and JPEG decoder for imaging. It will be shown that when the above guidelines are followed, the best performance will be achieved.

Contents

List of Figures

List of Tables

1 Introduction

The C5501 and C5502 were designed as low cost solutions for low cost real time systems. In order to reduce system cost and preserve the performance of the processor, the C5501 and C5502 DSPs employ an on-chip16K-byte instruction cache (I-Cache).

This application report was written to help the system designer achieve the best performance with this I-Cache memory architecture. For most applications this will require:

- Executing code in external memory and enabling the I-Cache
- Locating data in internal memory at run time

- Using the External Memory Interface (EMIF) at the largest possible clock frequency

C5502 benchmark results using these guidelines will be shown for several algorithms that are popularly used in low-cost systems. These applications were also chosen for their representation of code sizes relative to the size of the I-Cache.

- MP3 audio decoder is about 66% larger in size than the I-Cache.
- G.729AB vocoder is about 33% larger than the I-Cache.
- G.723.1 vocoder is about the same size as the I-Cache.
- JPEG imaging decoder is smaller than the I-Cache.

Although these benchmarks were run on the C5502, the guidelines for achieving best performance can still be applied to the C5501.

Please refer to the appendices in this document for further details on:

- Cache memory basics.
- Configuring the C5501/02 I-Cache using the Chip Support Library (CSL).
- C55x Cache Analysis Tool provided with Code Composer 2.20 simulators.
- Detailed throughput information for the C5501/02 EMIF using different types of memory.

2 Achieving the Best Performance with the C5502 DSP

The C5502 DSP has 32K 16-bit words of internal memory and the C5501 has half that size. In addition to their internal memories, both DSPs have a 16K-byte instruction cache. The following guidelines apply to applications that have memory requirements larger than the internal memory of these devices. These applications will require external memory.

2.1 Place Code in External Memory and Enable the I-Cache

For an application that requires external memory, place the code in external memory with the I-Cache enabled. The benchmarks provided in the next section show that executing code from the I-Cache will dramatically reduce or eliminate external memory access penalties.

The I-Cache can be enabled through an API provided in the C5501/02 Chip Support Library. Appendix B has more information regarding the configuration of the I-Cache.

2.2 Place Data in Internal Memory

In order to achieve best performance with the C5501/02, the data memory sections need to be allocated in internal memory.

If the data sections of the application are larger than size of the internal memory, then part of the data can be placed in external memory and transferred to internal memory when needed using the DMA channels.

2.3 Maximize the EMIF Clock Frequency

The C5501/02 EMIF has a specified maximum clock frequency of 100 MHz. The EMIF clock can be derived internally from the CPU clock or it can be provided from an external clock source. Furthermore, the CPU clock can be divided by one, two, or four to achieve the EMIF frequency.

When generating the EMIF clock internally, the smallest divider value that will generate the largest EMIF clock frequency should be used to divide the CPU clock. For example:

- When running the CPU at 300 MHz, a divider value of 4 should be used to generate the EMIF clock frequency from the CPU clock. In this case, the EMIF will run at 75MHz.
- When running the CPU at 200 MHz, a divider value of 2 should be used to generate the EMIF clock frequency from the CPU clock. In this case, the EMIF will run at 100MHz.

Details on the different clocking configurations for the EMIF are provided below.

The C5501 and C5502 have four clock groups: the C55x Subsystem Clock Group, the Fast Peripherals Clock Group, the Slow Peripherals Clock Group, and the External Memory Interface Clock Group. Clock groups allow for performance optimization and lower power since the frequency of groups with no high-speed requirements can be set to 1/4 or 1/2 the frequency of other groups.

The EMIF Clock Group controls the clock frequency of the External Memory Interface (EMIF) module. Divider 3 generates an input clock to the EMIF Clock Group by taking the clock of the C55x Subsystem Clock Group and dividing it by 1, 2, or 4 (see System Clock Generator section of the device specific data manual). By default, the divider is set to divide its input clock by four, but the divide value can be changed to divide-by-1 or divide-by-2 through the PLL Divider3 Register (PLLDIV3). Note that changing the input clock to the C55x Subsystem Clock Group by enabling the PLL or varying the input clock to the DSP will also affect the clock speed of the EMIF Clock Group.

The PLLDIV3 register should not be set in a manner that makes the frequency for the EMIF Clock Group greater than 100 MHz or greater than the frequency of the Fast Peripherals Clock Group, whichever is smaller. There must be no external memory accesses when the value of PLLDIV3 is being changed, this means that the frequency of the EMIF Clock Group cannot be changed by a program that is being executed from external memory. It is recommended that the EMIF be put in IDLE mode before changing the PLLDIV3 value.

The EMIF may also be clocked from an external asynchronous clock source through the ECLKIN pin if a specific EMIF frequency is needed. Although this feature provides system flexibility, the additional internal synchronization logic used can cause EMIF data throughput performance degradation. The source for the EMIF clock is selected by the EMIFCLKS pin. When this feature is not needed, the EMIFCLKS pin must be pulled low, enabling the internal clock described above with optimal throughput performance. If EMIFCLKS is high, the external asynchronous clock provided on the ECLKIN pin will be used as the input clock for the EMIF.

The external clock frequency can be up to 100 MHz. Also, when an external clock is specified, the PLLDIV3 register should be programmed to generate a clock with a frequency equal to or lower than that of the Fast Peripherals Clock Group.

3 Benchmarks

Several benchmarks were done on the C5502 DSP using several algorithms that are popularly used in low-cost systems. These applications were also chosen for their representation of code sizes relative to the size of the I-Cache. The benchmark results compare the performance of the C5502 with its instruction cache enabled and disabled using the C5510 DSP as a reference.

Although these benchmarks were run on the C5502, similar results can be obtained on the C5501 by following the guideliens outlined in Section 2.

3.1 Setup

The following setup was used to measure the performance impact of the I-Cache:

- A test board with external memory SDRAM and a C5502 DSP
- C5000 Code Composer Studio v2.20

The C5502 CPU was operated at 200 MHz and the EMIF module was set for the internal clocking option with the frequency set to divide by two of the CPU clock. The external SDRAM thus operated at 100 MHz.

3.2 G.723.1 Vocoder

The G.723.1 vocoder is used to compress voice in many telecommunication applications. The implementation that was used for this application report has the following memory requirements:

Table 1. G.723.1 Memory Requirement

	Program Source Code (Bytes)	Program XDAIS Wrapper (Bytes)	Scratch Memory (Words)	Persistent Memory/Channel (Words)	Table (Const) (Words)	Stack Memory (Words)
Encoder	16114	4546	2103	850	10672	70
Decoder	4306	2530	1413	283	10672	65
Full Duplex	18856	7076	2103	1133	10672	70

The “scratch” memory is the memory that is freely used by the algorithm without regard to its previous content. The “persistent” memory is used to store state information while an algorithm instance is not executing.

The input data is provided to the algorithm through the FILE I/O capabilities of Code Composer Studio. The data is read from a file and transferred into the input data buffers of the algorithm. The latency of this transfer is not taken into account in these benchmarks. Only the encode function was benchmarked.

Table 2. G.723.1 Encoder Benchmark

Benchmark	DSP	Internal Memory	External Memory	I-Cache Enabled	Cycles	Relative Performance
1	C5510	PROG + DATA	N/A	N/A	262000	Reference
2	C5502	DATA	PROG	No	1800000	7x
3	C5502	DATA	PROG	Yes	259000	1x

NOTES: 1. The cycle counts shown in this table represent an average over 200 frames of speech signal.
2. PROG: Program memory, DATA: Scratch + Persistent + Table + Stack memory.

In order to establish a reference for these benchmarks the code and data were placed in internal memory on a C5510 DSP. The cycle count measured for this configuration was used as a reference for the C5502 benchmarks. In benchmark 2, the code was placed in external memory and it was executed on a C5502 but the I-Cache was not turned on. A sharp degradation in the performance of the application was noticed. In benchmark 3, the same configuration was used as in benchmark 2 except that the I-Cache was enabled. The performance of the application with the code placed in external memory and the I-Cache enabled on the C5502 is similar to the performance with the code placed in internal memory on the C5510. Note that the cycle counts shown in Table 2, Table 4, and Table 6 represent an average of several data frames. The cycle count for the first frame will be higher because of I-Cache compulsory misses (compulsory misses are defined in Appendix A).

3.3 G.729AB Vocoder

The G.729AB vocoder is used to compress voice in applications such as Voice over Packet. The implementation that was used for this application report has the following memory requirements:

Table 3. G.729AB Memory Requirement

	Program Source Code (Bytes)	Program XDAIS Wrapper (Bytes)	Scratch Memory (Words)	Persistent Memory/Channel (Words)	Table (Const) (Words)	Stack Memory (Words)
Encoder	21246	1742	1114	1062	3171	124
Decoder	9792	1452	1114	702	3171	64
Full Duplex	31038	3194	1114	1764	3171	124

The “scratch” memory is the memory that is freely used by the algorithm without regard to its previous content. The “persistent” memory is used to store state information while an algorithm instance is not executing.

The size of the Encoder code is larger than the size of the I-Cache (16 Kbytes). The input data is provided to the algorithm through the FILE I/O capabilities of Code Composer Studio. The data is read from a file and transferred into the input data buffers of the algorithm. The latency of this transfer is not taken into account in these benchmarks. Only the encode function was benchmarked.

Table 4. G.729AB Encoder Benchmark

Benchmark	DSP	Internal Memory	External Memory	I-Cache Enabled	Cycles	Relative Performance
1	C5510	PROG + DATA	N/A	N/A	41000	Reference
2	C5502	DATA	PROG	No	472000	11.5x
3	C5502	DATA	PROG	Yes	50500	1.23x

NOTES: 1. The cycle counts shown in this table represent an average over 200 frames of speech signal.
2. PROG: Program memory, DATA: Scratch + Persistent + Table + Stack memory.

Benchmark 1 was obtained on the C5510 with Program and Data running out of internal memory. It is used as a reference. Benchmark 2 was run on the C5502 with code in external memory and I-Cache off. Benchmark 3 had the C5502 I-Cache enabled. The difference between Benchmarks 1 and 3 results from a code size that is larger than the size of the C5502 I-Cache. However, the difference is much less with I-Cache enabled vs. turned off showing the importance of the I-Cache for larger code sizes as well.

3.4 MP3 Decoder

The MP3 decoder is used to decompress audio in applications such as streaming audio. The implementation that was used for this application report has the following memory requirements:

Table 5. MP3 Decoder Memory Requirement

	Program Source Code (Bytes)	Program XDAIS Wrapper (Bytes)	Scratch Memory (Words)	Persistent Memory/Channel (Words)	Table (Const) (Words)	Stack Memory (Words)
Decoder	19600	9800	2304	5902	7000	100

The “scratch” memory is the memory that is freely used by the algorithm without regard to its previous content. The “persistent” memory is used to store state information while an algorithm instance is not executing.

The Decoder code is larger than the size of the I-Cache (16 Kbytes) and serves as a good benchmark for the benefits of the I-Cache. The input data is provided to the algorithm through the FILE I/O capabilities of Code Composer Studio. The data is read from a file and transferred into the input data buffers of the algorithm. The latency of this transfer is not taken into account in these benchmarks. Only the decode function was benchmarked.

Table 6. MP3 Decoder Benchmark

Benchmark	DSP	Internal Memory	External Memory	I-Cache Enabled	Cycles	Relative Performance
1	C5510	PROG + DATA	N/A	N/A	196,521	Reference
2	C5502	DATA	PROG	No	4,793,413	24.4x
3	C5502	DATA	PROG	Yes	205,794	1.047x

NOTES: 1. The cycle counts shown in this table represent an average over 200 frames of audio signal.
2. PROG: Program memory, DATA: Scratch + Persistent + Table + Stack memory.

In order to establish a reference for these benchmarks the code and data were placed in internal memory on a C5510 DSP. In benchmark 2 the code was placed in external memory of the C5502 and it was run with the I-Cache turned off. Benchmark 2 shows a sharp degradation in the performance of the application. In benchmark 3, the same configuration was used as in benchmark 2 except that the I-Cache was enabled resulting in much better performance for the C5502.

Benchmark 3 shows a degradation of 4.7% in the performance for the MP3 decoder when run from external memory with the I-Cache enabled. This degradation is explained by the fact that the code size is larger than the size of the I-Cache (+66%). However the degradation is smaller than expected because the code is not 100% covered by the I-Cache during the processing of these frames. The Code Composer Studio 2.2 Cache simulator was used to comprehend the results obtained in this benchmark. The results for that exercise are shown in Appendix C.

In conclusion, these benchmarks show that it is possible to run the MP3 decoder from external memory with the I-Cache enabled without significant performance degradation.

3.5 JPEG Decoder

JPEG is an international standard for color image compression. This standard is defined in the ISO 10918-1 JPEG Draft International Standard | CCITT Recommendation T.81. The C55x JPEG decoder is made of high-level C functional calls for ease of understanding processing flow and optimized assembly routines for high MIPS data processing. The implementation that was used for this application report has the following memory requirements:

Table 7. JPEG Decoder Memory Requirement

	Program Source Code (Bytes)	Program XDAIS Wrapper (Bytes)	Scratch Memory (Words)	Persistent Memory/Channel (Words)	Table (Const) (Words)	Stack Memory (Words)
Decoder	4918	292	2312	N/A	192	1702

The “scratch” memory is the memory that is freely used by the algorithm without regard to its previous content. The “persistent” memory is used to store state information while an algorithm instance is not executing.

In a real-world application, the JPEG Decoder is often integrated with a JPEG encoder and some audio codecs. Therefore, the program is located in external memory.

The input data is provided to the algorithm by a JPEG file. The data which is read from the JPEG file fills into the input data buffers of the algorithm. The latency of the file read is not taken into account in these benchmarks. Only the decoder function was benchmarked.

Table 8. JPEG Decoder Benchmark

Benchmark	DSP	Internal Memory	External Memory	I-Cache Enabled	Cycles	Relative Performance
1	C5510	PROG + DATA	N/A	N/A	2,686,064	Reference
2	C5502	DATA	PROG	No	9,252,994	3.45x
3	C5502	DATA	PROG	Yes	2,710,586	1.01x

NOTES: 1. The cycle counts shown represent the total number of cycles needed to process one image of format QCIF YUV4:2:0.
2. PROG: Program memory, DATA: Scratch + Persistent + Table + Stack memory.

When the code is placed in the external memory used by the C5502 and the I-Cache is enabled, performance is near equivalent to the same code and data running out of internal memory on the C5510. This degradation is only caused by the compulsory misses because the code size is smaller than the cache size (compulsory misses are defined in Appendix A). The performance impact by the compulsory misses is usually small. Notice that the performance deteriorates when the I-Cache is not enabled.

When the JPEG decoder is integrated with other algorithms such as JPEG encoder and audio codec, these benchmarks show that in order to get the best performance of the JPEG decoder running on a C5502 DSP, the data memory sections should be placed in internal memory (RAM or ROM for the tables) and the program memory sections should be placed in external memory with the I-Cache enabled.

4 Conclusion

The C5501 and C5502 DSPs were designed as low-cost DSPs for low-cost solutions. In order to achieve the best performance with this processor, the system designer must follow the guidelines provided in the application report.

- Code placed in external memory and I-Cache enabled
- Data located in internal memory at run time
- External Memory Interface (EMIF) maximizes clock frequency

The examples provided in this application report have shown that when these guidelines are followed the performance is often comparable to the C5510 executing code from internal memory.

5 References

1. *TMS320VC5502 Fixed-Point Digital Signal Processor Data Manual* (SPRS166)
2. *TMS320VC5501/5502 DSP Instruction Cache Reference Guide* (SPRU630)
3. *TMS320VC5501/5502 DSP External Memory Interface (EMIF) Reference Guide* (SPRU621)
4. *TMS320C55x Chip Support Library Reference Guide* (SPRU433)
5. *Cache Analysis User's Guide* (SPRU575)

Appendix A Cache Basics

Memory consumption for real-time algorithms covers a wide spectrum. In order to reduce the memory costs of a system while at the same time managing processing bottlenecks, cache-based architectures have been developed. Caches are small, fast memory that reside between the CPU and the slower external memory. The cache provides code to the CPU at the speed of the processor while automatically managing the data movement from the slower main memory that is frequently located off-chip.

This section will introduce the basic conceptual ideas behind cache, though many abstractions are used for simplification. Please refer to *Cache Analysis User's Guide* (SPRU575) for a more detailed discussion.

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

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

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

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

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

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

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

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

These sources of misses can be reduced by a number of code optimization techniques. Conflict misses can be eliminated by changing the locations of data or program code in memory, and hence they will not contend for the same cache line. Capacity misses can be reduced by working on smaller amounts of data or code at a time, which can be achieved by reordering the accesses of the data or by partitioning the algorithm into smaller pieces. Please refer to *Cache Analysis User's Guide* (SPRU575) for more discussions on cache optimization techniques.

Appendix B The C5501/02 Instruction Cache

B.1 Instruction Cache Description

The I-Cache implemented in the C5501 and C5502 is a 2-way set associative cache that holds up to 16K bytes: 512 sets, 2 lines per set, four 32-bit words per line. The I-Cache is controlled through three bits which are included in the ST3_55 status register. The three bits can enable, freeze and flush (invalidate all the lines) the I-Cache. The flush operation is defined as the invalidation of all the lines of the I-Cache. For more information about the C5501/02 I-Cache please refer to the *TMS320VC5501/5502 DSP Instruction Cache Reference Guide* (SPRU630)**.**

B.2 Instruction Cache Operation

In the C5501/02 architecture, the program code is being fetched in 32-bit packets. When the program code is located in internal memory, the I-Cache plays no role in the fetching of the code packets. When the program code resides in external memory and the I-Cache is enabled, the fetch requests are sent by the CPU to the I-Cache. If the requested 32-bit packet is not available in the I-Cache, the I-Cache will fetch the 128-bit packet that contains the 32-bit packet through the external memory interface (EMIF) from external memory. After fetching the 128-bit packet from external memory it will forward the 32-bit packet to the CPU. This operation is transparent to the user as long as the I-Cache is enabled. If the I-Cache is not enabled the CPU request will be sent directly to the EMIF and the 32-bit packet will be fetched from external memory and sent directly to the CPU.

B.2.1 Initialization Steps

1. Write 0xCE3C to the I-Cache Global Control (ICGC) register.
2. Set the cache enable bit (CAEN) of the status register ST3_55 to one.

B.2.2 Configuration

The user can configure the I-Cache in the following ways using the three control bits CAEN, CACLR, CAFRZ located in ST3_55:

- Enable I-Cache:

  Perform as part of the initialization.

- Disable I-Cache:

  Clear the cache enable bit (CAEN) of the status register ST3_55 to zero.

  Note: When the I-Cache is disabled the content of the I-Cache is not invalidated. This means that the I-Cache can be re-enabled if needed and the content will be the same.

- Flush (invalidate) content of I-Cache

  Set the I-Cache flush bit (CACLR) of the status register ST3_55 to one.

  Note: Upon completion of the flush (invalidate) process the CACLR bit is automatically reset to zero.

- Freeze content of I-Cache

  Set the cache freeze bit (CAFRZ) of the status register ST3_55 to one.

Note: When the content of the I-Cache is frozen, the lines are not updated in response to an I-Cache miss. However, the miss still produces a full 16-byte fetch in the EMIF. Therefore, when freezing the cache the code has to be carefully profiled in order to evaluate the performance impact.

- Un-freeze content of I-Cache

  Clear the cache freeze bit (CAFRZ) of the status register ST3_55 to zero.

B.3 Using the Chip Support Library (CSL) to Configure the I-Cache

The CSL is a software library that was developed to configure peripherals available on the Texas Instruments DSPs. The Code Composer Studio v.2.20 provides a CSL library for the C5502 DSP. This library contains APIs to configure the I-Cache. For more information about these APIs please refer to the *TMS320C55x Chip Support Library Reference Guide (SPRU433).*

The following code shows how to use the I-Cache APIs to configure the C5501/02 I-Cache.

```
void main()
{
       int j;
       int x = 0;

       CSL_init();             // MANDATORY, initialized CSL

       I-CACHE_setup();        // Initialize and enable I-Cache

       function1();

       I-CACHE_flush();        // Flush (invalidate) the I-Cache

       function2();


       I-CACHE_freeze();       // Freeze the I-Cache

       function3();

       I-CACHE_unfreeze();     // Un-freeze the I-Cache

       function2();

       I-CACHE_disable();      // Disable the I-Cache

       function4();

       I-CACHE_enable();       // Enable the I-Cache

       function2();

}
```

Appendix C The 55x Cache Analysis Simulator

C.1 Introduction to Cache Analysis Tools

TI’s Cache Analysis tool is aimed at identifying cache efficiency problem areas and visualizing them in a way to facilitate making rapid improvements in cache performance. The Cache Analysis tool is based on the simulation platform. The C55x cache simulator is capable of generating cache trace file (i.tip) that is visualized using this tool. The trace files can be viewed inside the two-dimensional address-over-time display grid. All the accesses are color-coded by type (Miss = red, hit = green, etc.) and various filters, panning, and zoom features facilitate quick drill down. This visual temporal view of the cache accesses enables quick identification of problem areas, and whether they are conflict, capacity, or compulsory related. All of these features combined greatly ease efforts to analyze cache activity

The following sequence illustrates the steps needed with each phase of the development flow. It is based on the assumption that the reader is already familiar with Code Composer Studio IDE and how to setup Code Composer Studio as well as how to create and load projects using Code Composer Studio. Some simplifications are used for brevity. Please refer to the *Cache Analysis User’s Guide* (SPRU575) for detailed descriptions on the features and discussion on cache optimization techniques.

C.2 Using the Cache Simulator to Analyze the Performance of Code

Step A: Set Up the C55x Cache Simulator

The first step is to set up the C55x Cache simulator, which gives accurate program execution time while simulating I-Cache. It is assumed the reader is familiar with the Code Composer Studio setup interface

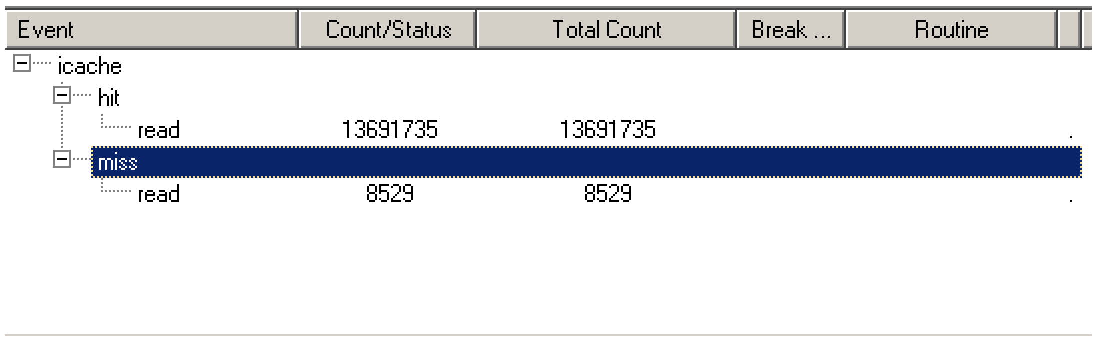
1. Start the Code Composer Studio setup utility.
2. From the list of available configurations within the Import Configuration dialog box, select the standard configuration C55x Cache Simulator as the target simulator.
3. In Board Properties window, change the Simulator Config File from the default SIM55xx_csim.cfg to SIM55xx_csim_profile.cfg.
4. Save the configuration and exit the setup utility.

Step B: Load the Project (MP3 Decoder) in Code Composer Studio

Step C: Profile

The reader will use Simulator Analysis plug-in to gather cache statistics. Please refer to Code Composer Studio online help for detailed information on the plug-in.

1. Choose Tools → Simulator Analysis to invoke Simulator Analysis plug-in, which reports the particular system events so that we can accurately monitor and measure the performance of our program.
2. In Analysis Setup window, enable the analysis and setup the events to count. The events we are interested in are cache events. Select the events for I-Cache hits and misses. The events selected will be displayed in Simulator Analysis Window.
3. Run the application to completion. The statistics of selected events will be displayed in the Simulator Analysis Window. A screen shot is shown in Figure C−1.



Event	Count/Status	Total Count	Break ...	Routine
icache				
hit				
read	13691735	13691735		
miss				
read	8529	8529		

Figure C−1. MP3 I-Cache hit/miss (running in C55x cache simulator)

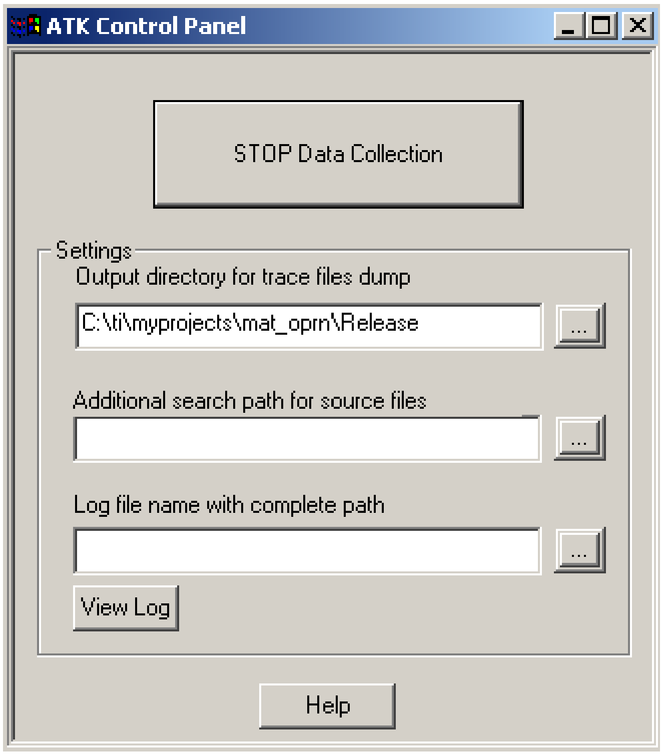
Figure C−1 shows that the number of I-Cache hits is more than 1600 times of the number of I-Cache misses. This explains why the benchmark 1 and benchmark 3 are so close in Table 6. If the total number of misses had been greater, then benchmark 3 would not have shown such a comparable performance to benchmark 1.

The number cache misses depends on factors such as the number of functions that are called, the size of the called functions, the order in which the functions are called, the number of times each function is called, etc. For example, for a particular algorithm, say there were small a number of functions used for initialization and a few functions that were used very frequently yet were small enough to fit inside the cache. If the initialization functions were not called very often, then the number misses generated by calling them would have minimal impact on the total cycle count. On the other hand if the frequently used functions did not fit entirely in the cache, they would generate a lot of misses, increasing the total cycle time needed to run an algorithm, which would in turn decrease its performance when compared to algorithms running entirely from internal memory.

Providing specific details as described above for the MP3 algorithm would be very lengthy and beyond the scope of this application note. The purpose here is to show that the performance is very comparable to executing from internal memory when the hit-to-miss ratio is very large.

Step D: Invoke ATK Control Panel to Generate Trace Files

1. Reload MP3 decoder project.
2. Invoke ATK Control Panel from menu Tools → Analysis Toolkit. A screen shot of the control panel is shown in Figure C−2.



Settings

Output directory for trace files dump: C:\ti\myprojects\mat_oprn\Release

Additional search path for source files

Log file name with complete path

Figure C−2. ATK Control Panel

3. Specify the output directory for trace file dump. By default, the trace files are generated into the directory from which the program was loaded.
4. Run the application to completion. When the application executable is loaded in step 1, data collection implicitly starts. The data collection also implicitly stops when the program stops. Now the trace data have been collected and we are ready to launch the Cache Analysis tool.
5. Choosing Tools → Analysis Toolkit → Cache Analysis to launch Cache Analysis tool. By default, it will open the i.tip file. The cache activity will be shown when the i.tip file is opened.

Appendix D C5501/02 EMIF Throughput Analysis

D.1 Analyzing EMIF Throughput

The C5501/02 EMIF throughput for several types of memory accesses was measured using three different types of memory: synchronous dynamic random-access memory (SDRAM), synchronous burst random-access memory (SBSRAM), and asynchronous memory. Code was written to perform specific memory access operations and the word access time for each operation was measured in terms of EMIF clock cycles. For example, code was written to have the CPU perform consecutive data reads from SDRAM and executed from both internal and external memory to measure the access time for the read operation. Cycle access times were measured for both CPU and DMA memory accesses.

All measurements were conducted with code executing from both internal and external memory. When executing code from external memory, the cycle access time was measured during different states of the instruction cache.

- **Cache disabled.** The cycle access time for the operation under question was measured while code was being executed from external memory. In this case, the total data access cycle time includes the number of cycles to access the data as well as the number of cycles needed to read application code from external memory.
- **Cache filling.** The cycle access time for the operation under question was measured during a cache miss condition. The cache requests a 128-bit packet from external memory to fill one its lines during a cache miss. Since the external memory is 32-bits wide, the EMIF will service the 128-bit packet request from the instruction cache as a read burst of four 32-bit words. In this case, the total data access cycle time includes the number of cycles to accesses the data as well as the number of cycles needed to read 4 32-bit words of application code from external memory.
- **Cache filled.** The cycle access time for the operation under question was measured with all code executing directly from the instruction cache. In this case, all application code is running straight from the instruction cache, therefore, there are not application code reads to affect the total data access cycle time.

The following memory devices were used for all of the experiments presented in this appendix:

- Micron MT48LC2M32B2 64Mbit (2Mx32) SDRAM
- Micron MT58L256L32PS SBSRAM
- Toshiba TC55V16256FT-12 Asynchronous Memory

The following EMIF configuration was used for each type of memory:

- SDRAM
  - TRC = 5
  - TCL = 1 (CAS latency = 3 ECLKOUT1 cycles)
  - SDWTH[4] = 1b (4 banks)
  - SDWTH[3:2] = 00b (11 row address pins)
  - SDWTH[1:0] = 01b (8 column address pins)

  - TRCD = 1
  - TRP = 1
  - MTYPE = 0011b (32-bit SDRAM)
- SBSRAM
  - SYNCRL = 10b (2 cycle read latency)
  - SYNCWL = 00b (0 cycle write latency)
  - SNCCLK = 0b (control/data signals for this CE space are synced to ECLKOUT1)
  - MTYPE = 0100b (32-bit SBSRAM)
- Asynchronous Memory
  - READ/WRITE SETUP = 2
  - READ/WRITE STROBE = 2
  - READ/WRITE HOLD = 2
  - MTYPE = 0010b (32-bit Asynchronous Memory)

All the parameters presented above are described in more detail in the *TMS320VC5501/5502 DSP External Memory Interface (EMIF) Reference Guide* (SPRU621).

The EMIF was operated at 100 MHz while the CPU operated at 200 MHz.

D.2 CPU External Memory Accesses

The EMIF throughput for CPU external memory accesses was analyzed by measuring the number of EMIF clock cycles for several CPU memory access operations. For each case, the operation was repeated and the average EMIF cycle time was measured for that operation. The results for these measurements are given in the tables below under the “Program Internal” column.

The same tests were repeated with the code executing from external memory. The EMIF throughput for CPU memory accesses was measured with the cache disabled, during a cache miss, and with the program executing totally from within the instruction cache. The total cycle time in this case includes the number of cycles needed for the data access plus the number of cycles needed for instruction fetches (in the case of cache disabled) or line fill requests from the cache (in the case of a cache miss). All results for these measurements are given in the tables below under the “Program External” column.

Note that the total number of cycles given for the cache disabled condition depends on the size of the instruction. The total number of cycles for the chase miss condition would not be dependant on the size of the instruction since the cache will always request 4 32-bit words to do a line fill.

The number of EMIF clock cycles needed for an instruction fetch was also measured for all three memory types. Notice that when the instruction cache is disabled, 32 bits of code are fetched from program space and sent to the instruction buffer unit. On the other hand, when the instruction cache is enabled, four 32-bit words are fetched to fill one line in the instruction cache. The information for these two operations is included in the tables below. Refer to the TMS320C55x DSP CPU Reference Guide (SPRU371) for more information on the instruction buffer unit.

All results below show the number of EMIF clock cycles for 32-bit word accesses (words/EMIF clock cycles).

Table D−1. EMIF Throughput for CPU Accesses (SDRAM)†

CPU Operation	Program Internal	Program External
Instruction Fetch		With cache disabled: 1/18 With cache enabled: 4/27
Read after Read	1/18	With cache disabled: 1/40 While cache fills: 1/44 With cache filled: 1/18
Write after Write‡	With WP: 1/9 Without WP: 1/11	With cache disabled: 1/25 With cache filled: 1/31 With cache filled, WP: 1/11 With cache filled, w/o WP: 1/9
Read after Write (read cycle)	1/19	With cache disabled: 1/35 While cache fills: 1/41 With cache filled: 1/20
Write after Read (write cycle)	1/10	With cache disabled: 1/42 While cache fills: 1/10 With cache filled: 1/10

† Results show number of 32-bit words per EMIF clock cycles (words/cycles).
‡ WP = write-posting. Note that write-posting only affects cycle counts on back-to-back write operations. Cache conditions such as cache disabled and cache filling will prevent any write posting from taking place since consecutive write operations will be interrupted by application code read operations.

Table D−2. EMIF Throughput for CPU Accesses (SBSRAM)†

CPU Operation	Program Internal	Program External
Instruction Fetch		With cache disabled: 1/15 With cache enabled: 4/21
Read after Read	1/15	With cache disabled: 1/28 While cache fills: 1/34 With cache filled: 1/15
Write after Write‡	With WP: 1/9 Without WP: 1/11	With cache disabled: 1/22 While cache fills: 1/28 With cache filled, WP: 1/9 With cache filled, w/o WP: 1/11
Read after Write (read cycle)	1/17	With cache disabled: 1/28 While cache fills: 1/34 With cache filled: 1/17
Write after Read (write cycle)	1/9	With cache disabled: 1/22 While cache fills: 1/28 With cache filled: 1/9

† Results show number of 32-bit words per EMIF clock cycles (words/cycles).
‡ WP = write-posting. Note that write-posting only affects cycle counts on back-to-back write operations. Cache conditions such as cache disabled and cache filling will prevent any write posting from taking place since consecutive write operations will be interrupted by application code read operations.

Table D−3. EMIF Throughput for CPU Accesses (Asynchronous Memory)[†]

CPU Operation	Program Internal	Program External
Instruction Fetch		With cache disabled: 1/16 With cache enabled: 4/41
Read after Read	1/16	With cache disabled: 1/27 While cache fills: 1/53 With cache filled: 1/16
Write after Write[‡]	With WP: 1/9 Without WP: 1/11	With cache disabled: 1/20 While cache fills: 1/41 With cache filled, WP: 1/9 With cache filled, w/o WP: 1/11
Read after Write (read cycle)	1/18	With cache disabled: 1/41 While cache fills: 1/55 With cache filled: 1/18
Write after Read (write cycle)	1/9	With cache disabled: 1/9 While cache fills: 1/9 With cache filled: 1/9

[†] Results show number of 32-bit words per EMIF clock cycles (words/cycles).
[‡] WP = write-posting. Note that write-posting only affects cycle counts on back-to-back write operations. Cache conditions such as cache disabled and cache filling will prevent any write posting from taking place since consecutive write operations will be interrupted by application code read operations.

D.3 DMA External Memory Accesses

The EMIF throughput for DMA external memory accesses was analyzed by measuring the number of EMIF clock cycles for several DMA memory accesses operations. The results for these measurements are given in the tables below under the “Program Internal” column.

Note that, for all the measurements included in these tables, no DMA synchronization events were used and the DMA was programmed to access 32-bit words from a single EMIF CE space in bursts of four.

The impact of executing code from external memory on the EMIF throughput was analyzed by having the CPU execute dummy 32-bit instructions while the DMA performed different external memory accesses. The EMIF throughput was measured with the cache disabled, during a cache miss, and with the program executing totally from within the instruction cache. The total cycle time in this case includes the number of cycles needed for the data access plus the number of cycles needed for instruction fetches (in the case of cache disabled) or the line fill requests from the cache (in the case of a cache miss). All results for these measurements are given in the tables below under the “Program External” column.

Note that the total number of cycles given for the cache disabled condition depends on the size of the instruction being fetched from external memory; the number given in the tables below is for the worst case scenario of a 6-byte instruction. The total number of cycles for the chase miss condition would not be dependant on the size of the instruction since the cache will always request 4 32-bit words to do a line fill.

All results below show the number of EMIF clock cycles for 32-bit word accesses (words / EMIF clock cycles). Keep in mind that the DMA has been configured to perform memory accesses in burst of four.

Table D−4. EMIF Throughput for DMA Accesses (SDRAM)[†]

DMA Operation	Program Internal	Program External
Write after Write (1 Channel)	4/21	With cache disabled: 4/40 While cache fills: 4/46 With cache filled: 4/21
Write after Write (2 Channels)	4/18	With cache disabled: 4/46 While cache fills: 4/46 With cache filled: 4/21
Read after Read (1 Channel)	4/28	With cache disabled: 4/46 While cache fills: 4/52 With cache filled: 4/28
Read after Read (2 Channels)	4/27	With cache disabled: 4/46 While cache fills: 4/52 With cache filled: 4/27
Read plus Write (2 Channels)[‡]	(4 read + 4 write)/48	With cache disabled: (4 read + 4 write)/80 While cache fills: (4 read + 4 write)/92 With cache filled: (4 read + 4 write)/48
Write after Write (1 channel)[§]	(4 read + 4 write)/52	With cache disabled: (4 read + 4 write)/80 While cache fills: (4 read + 4 write)/98 With cache filled: (4 read + 4 write)/52

[†] Results show number of 32-bit words per EMIF clock cycles (words/cycles).
[‡] One channel transfers data from internal to external memory (write) while the other transfers data from external to internal memory (read).
[§] External to external memory DMA transfers. Note that for this memory access, a read must be performed to fetch the data from external memory before the write to external memory can take place.

Table D−5. EMIF Throughput for DMA Accesses (SBSRAM)†

DMA Operation	Program Internal	Program External
Write after Write (1 Channel)	4/20	With cache disabled: 4/30 While cache fills: 4/36 With cache filled: 4/20
Write after Write (2 Channels)	4/18	With cache disabled: 4/30 While cache fills: 4/36 With cache filled: 4/18
Read after Read (1 Channel)	4/25	With cache disabled: 4/34 While cache fills: 4/34 With cache filled: 4/25
Read after Read (2 Channels)	4/21	With cache disabled: 4/34 While cache fills: 4/34 With cache filled: 4/21
Read plus Write (2 Channels)‡	(4 read + 4 write)/38	With cache disabled: (4 read + 4 write)/64 While cache fills: (4 read + 4 write)/76 With cache filled: (4 read + 4 write)/38
Write after Write (1 channel)§	(4 read + 4 write)/42	With cache disabled: (4 read + 4 write)/64 While cache fills: (4 read + 4 write)/76 With cache filled: (4 read + 4 write)/42

† Results show number of 32-bit words per EMIF clock cycles (words/cycles).
‡ One channel transfers data from internal to external memory (write) while the other transfers data from external to internal memory (read).
§ External to external memory DMA transfers. Note that for this memory access, a read must be performed to fetch the data from external memory before the write to external memory can take place.

Table D−6. EMIF Throughput for DMA Accesses (Asynchronous Memory)†

DMA Operation	Program Internal	Program External
Write after Write (1 Channel)	4/29	With cache disabled: 4/43 While cache fills: 4/55 With cache filled: 4/29
Write after Write (2 Channels)	4/29	With cache disabled: 4/43 While cache fills: 4/56 With cache filled: 4/29
Read after Read (1 Channel)	4/41	With cache disabled: 4/48 While cache fills: 1/48 With cache filled: 4/41
Read after Read (2 Channels)	4/37	With cache disabled: 4/48 While cache fills: 4/60 With cache filled: 4/37
Read plus Write (2 Channels)‡	(4 read + 4 write)/68	With cache disabled: (4 read + 4 write)/91 While cache fills: (4 read + 4 write)/115 With cache filled: (4 read + 4 write)/68
Write after Write (1 channel)§	(4 read + 4 write)/72	With cache disabled: (4 read + 4 write)/91 While cache fills: (4 read + 4 write)/96 With cache filled: (4 read + 4 write)/72

† Results show number of 32-bit words per EMIF clock cycles (words/cycles).
‡ One channel transfers data from internal to external memory (write) while the other transfers data from external to internal memory (read).
§ External to external memory DMA transfers. Note that for this memory access, a read must be performed to fetch the data from external memory before the write to external memory can take place.

D.4 EMIF Throughput Comparison to C5510 for SDRAM Accesses

The tables below compare some of the C5501/02 EMIF throughput measurements to those available for the TMS320VC5510. The measurements being compared are for CPU and DMA accesses from SDRAM with the program executing from internal memory. The EMIF setup and test setup was the same for both DSPs.

Table D−7. C5502 EMIF Throughput Comparison to C5510 EMIF for CPU Accesses (SDRAM)†

CPU Operation	C5510 Program Internal	C5501/02 Program Internal
Read after Read	1/4	1/18
Write after Write‡	With WP: 1/1 Without WP: 1/4	With WP: 1/9 Without WP: 1/11
Read after Write (read cycle)	1/9	1/19
Write after Read (write cycle)	1/6	1/10

† Results show number of 32-bit words per EMIF clock cycles (words/cycles).
‡ WP = write-posting. Note that write-posting only affects cycle counts on back-to-back write operations. Cache conditions such as cache disabled and cache filling will prevent any write posting from taking place since consecutive write operations will be interrupted by application code read operations.

Table D−8. C5502 EMIF Throughput Comparison to C5510 EMIF for DMA Accesses (SDRAM)[†]

DMA Operation	C5510 Program Internal	C5501/02 Program Internal
Write after Write (1 Channel)	8/13	4/21
Write after Write (2 Channels)	64/67	4/18
Read after Read (1 Channel)	8/30	4/28
Read after Read (2 Channels)	4/10	4/27

[†] Results show number of 32-bit words per EMIF clock cycles (words/cycles).

D.5 Calculating MegaBytes per Second (MBPS)

The throughput data presented in the previous sections can be used to calculate the number of bytes that can be accessed by the C5501/02 EMIF per second using the formula:

$$MBPS = (words/cycles) \times (cycles/second) \times (bytes/word)$$

For example, MBPS for a CPU read instruction from asynchronous memory can be calculated as follows (assume EMIF is running at 100MHz):

$$MBPS = (1\ word/13\ cycles) \times (100 \times 10^6\ cycles/second) \times (4\ bytes/word)$$

$$MBPS = 30.77\ Mbytes/sec$$

Note: Cycles in this case refers to EMIF clock cycles.

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