Application Report
SPRA705 – December 2000

Optimizing JPEG on the TMS320C6211 2-Level Cache DSP

Jungki Min, Vishal Markandey *Digital Signal Processing Solutions*

ABSTRACT

This application report describes the implementation and optimization techniques of the Joint Photographic Experts Group (JPEG), the still image compression standard on the TMS320C6211. The TMS320C6211 is a low-cost and high-performance digital signal processor (DSP) with 2-level caches, that utilizes the VelociTI™ very-long-instruction-word (VLIW) architecture.

The TMS320C6211 has 64KB of L2 memory which is configurable for the combination of cache and SRAM, and has 4 KB each of L1 instruction cache and L1 data cache.

This application report describes an overview of the JPEG standard and the optimization techniques used in the TI JPEG encoder and decoder. Unlike on the TMS320C6201 with relatively bigger internal memory, maximizing the cache ability is the key to boost the overall performance on the TMS320C6211. Several optimization techniques and ideas are used and they are described in the order of importance to the performance.

Keeping the L1 data cache from thrashing data, so that it utilizes the maximum capacity and associativity, is of primary importance. The corresponding performance on various levels of the L1 data cache usage has been tested and illustrated in this paper.Also shown is how to avoid or reduce the L1 instruction cache thrashing by re-aligning critical kernels inside the main loop.

This paper will provide pointers on how a developer can easily optimize existing algorithms from the TMS320C6201 or implement completely new algorithms on the TMS320C6211.

Contents

Figures

Tables

1 The JPEG (ISO DIS 10918) Standard

It is recommended that this application note be read with reference to the JPEG standards document ISO DIS 10918.

This application note does not describe the JPEG compression standard. It describes optimization ideas and techniques that were used to develop an efficient implementation of JPEG on the C6211 VelociTI architecture with 2 level caches. Frequent references are made to the standards document when referring to formats, tables, algorithms, mathematical expressions and examples.

This particular implementation satisfies the following requirements in the '**Baseline Process'** of the JPEG standard as described in the document **ISO DIS 10918-1 Requirements and Guidelines**.

- DCT-based, Sequential, 8-bit precision samples of image components (Y-Cb-Cr 4:4:4 / 4:2:2 / 4:2:0).
- 2 Quantization tables (one each for luma and chroma). Supports tables K1 and K2 in the standards document.
- 2 DC, 2 AC tables (separate sets for luma and chroma). Supports tables K3, K4, K5 and K6 in the standards document.
- Non-interleaved scans, one complete image component per scan.

2 Restrictions

The implementation in its present form imposes some restrictions on the format of the bit-stream. These can, however, be relaxed by appropriate changes in the main control logic. The restrictions are stated with reference to the JPEG standards document ISO DIS 10918.

- The **encoder** expects the image data as three separate raster scanned components for color images (i.e., **non-interleaved** Y-Cb-Cr) and one component for grayscale images.
- The **decoder** outputs the image as three separate raster scanned components in contiguous memory (Y-Cb-Cr) for color images and only Y component for grayscale images.
- Each scan contains a complete image component. A single image component is contained in a scan.
- The implementation does not handle 'restart intervals'.
- The **decoder** expects the **first scan** to start within the **first DMA packet** in the bit-stream. At least following markers should be included in the first DMA packet.
  - 0xFFD8: Start of Image
  - 0xFFC0: Start of Frame (Baseline DCT)
  - 0xFFDB: Define Quantization Table
  - 0xFFDA: Start of Scan

The TI JPEG encoder outputs those markers within the first 590 bytes, so the decoder minimum input DMA packet should be 590 bytes and the default is set to 640 bytes.

The TI JPEG decoder alone is not recommended to be used for decoding general JPEG images, since it was originally designed only for working together with the TI JPEG encoder.

3 eXpressDSP Algorithm Standard Compliancy

The purpose of the eXpressDSP Algorithm Standard is to reduce the factors that prohibit an algorithm to be easily integrated into a system without significant reengineering by the system integrator [10] [11] [12]. All algorithms must comply with a generic resource management API, called IALG, which is the core interface to meet the algorithm standard requirements.

This implementation of TI JPEG is fully eXpressDSP compliant.

4 Encoder

4.1 Description

The JPEG standard is a broad standard encompassing several compression and transmission modes. In order to facilitate future expansion to other modes, this implementation has a very modular construction. A single thread of control code handles all individual routines (kernels) which are called multiple times as required by the application. This control code will always be in *'C'* to facilitate changes in control architecture.

The encoding process consists of several data processing and transmission operations. The encoder has to insert several headers (frame-header, scan-headers etc.) into the JPEG bit-stream to facilitate decoding. The standard specifies that a JPEG file contains all the necessary tables required for decoding. Hence, the encoder has to perform several auxiliary transmission related functions in addition to image compression.

The control function `jpgenc_ti( )` chronologically calls all the encoder component routines like DCT, quantization, run-level encoding, variable length encoding etc., and performs data translations between the encoder routines. It operates the required double buffering scheme for DMA read-in of image component data from the external memory and DMA write-out of the JPEG bitstream to the external memory. It takes care of parallelizing the DMA data transfers with the core encoding kernels.

Figure 1 shows the block diagram for the control flow of the encoder in its present implementation.

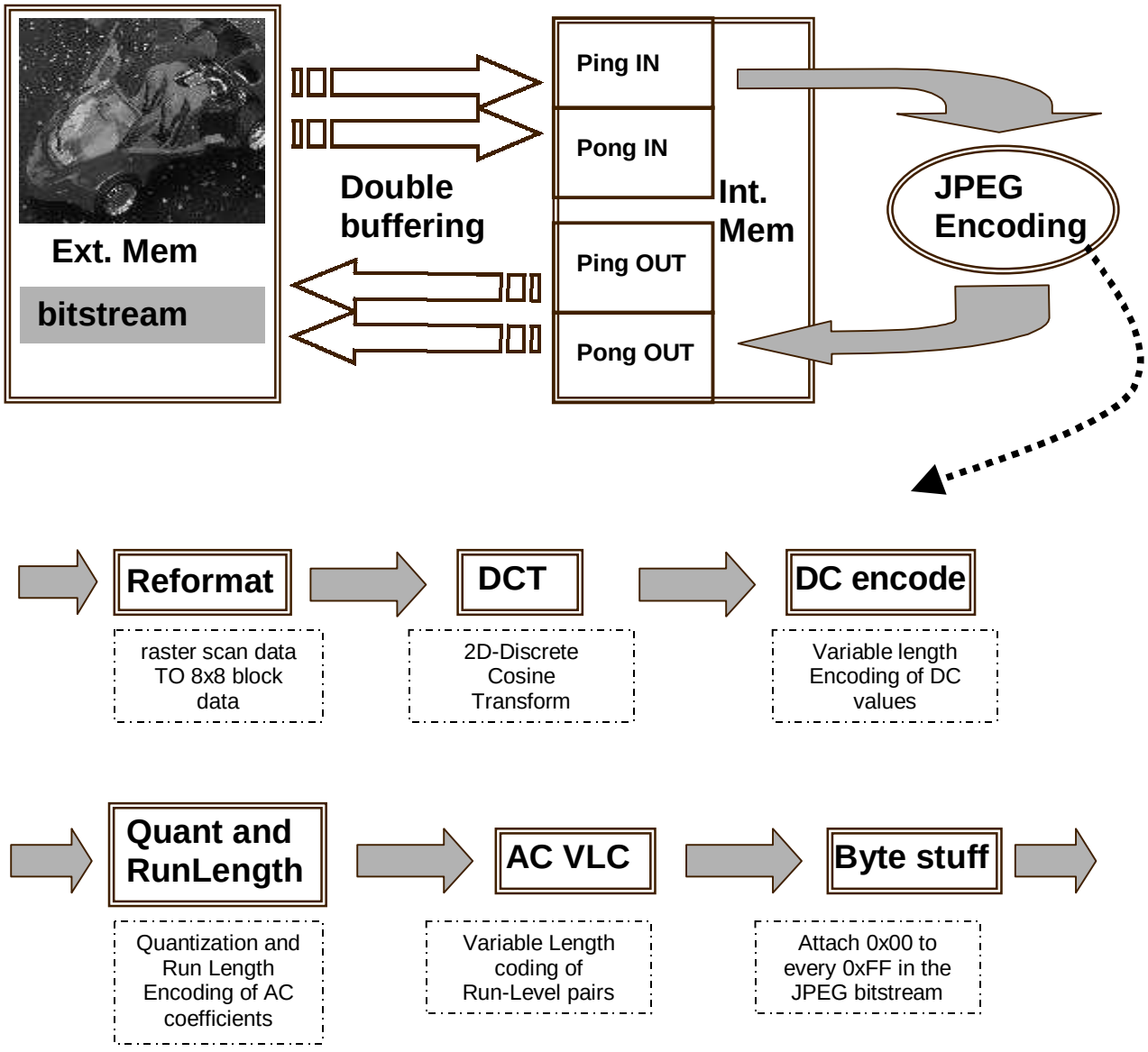

Figure 1. Block Diagram for the Control Flow of the Encoder

The driver should set up the parameters for the JPEG encoder by calling `JPEG_ENC_TI_control()` and execute the encoder by calling `JPEG_ENC_TI_encode()`. The driver should specify the configurations of the internal and external memories and set up the L2 cache operation and initialize the DMA by calling the corresponding CSL functions.

4.2 Encoder API

The API wrapper is derived from template material provided in the TMS320 DSP Algorithm Standard documentation. Knowledge of the algorithm standard is essential to understand the API wrapper. A complete discussion on how to make the algorithm eXpressDSP compliant is

beyond the scope of this document, however the algorithm interface will be discussed as knowledge of this ensures inter-operability of algorithms An algorithm is said to be eXpressDSP compliant if it implements the IALG Interface and observes all the programming rules in the algorithm standard. The core of the IALG interface is the IALG_Fxns structure type, in which a number of function pointers are defined. Each eXpress DSP-compliant algorithm **must** define and initialize a variable of type IALG_Fxns. In IALG_fxns, algAlloc(), algInit() and algFree() are required, while other functions are optional.

```
typedef struct IALG_Fxns {
    Void    *implementationId;
    Void    (*algActivate)(IALG_Handle);
    Int     (*algAlloc)(const IALG_Params *, struct IALG_Fxns **, IALG_MemRec *);
    Int     (*algControl)(IALG_Handle, IALG_Cmd, IALG_Status *);
    Void    (*algDeactivate)(IALG_Handle);
    Int     (*algFree)(IALG_Handle, IALG_MemRec *);
    Int     (*algInit)(IALG_Handle, const IALG_MemRec *, IALG_Handle, const
                       IALG_Params *);
    Void    (*algMoved)(IALG_Handle, const IALG_MemRec *, IALG_Handle, const
                        IALG_Params *);
    Int     (*algNumAlloc)(Void);
} IALG_Fxns;
```

The algorithm implements the algAlloc() function to inform the framework of its memory requirements by filling the memTab structure. It also informs the framework whether there is a parent object for this algorithm. Based on information it obtains by calling algAlloc(), the framework then allocates the requested memory. AlgInit() initializes the instance persistent memory requested in algAlloc(). After the framework has called algInit(), the instance of the algorithm pointed to by handle is ready to be used.

To delete an instance of the algorithm pointed to by handle, the framework needs to call algFree(). It is the responsibility of the algorithm to set the addresses and the size of each memory block requested in algAlloc() such that the application can delete the instance object without creating memory leaks.

The API for the JPEG Encoder is:

```
/*
 *  ======== ijpegenc.h ========
 *  IJPEGENC Interface Header
 */
#ifndef IJPEGENC_
#define IJPEGENC_

#include <std.h>
#include <xdas.h>
#include <ialg.h>
#include <ijpeg.h>

/*
 *  ======== IJPEGENC_Handle ========
 *  This handle is used to reference all JPEGENC instance objects
 */
typedef struct IJPEGENC_Obj *IJPEGENC_Handle;

/*
 *  ======== IJPEGENC_Obj ========
 *  This structure must be the first field of all JPEGENC instance objects
 */
typedef struct IJPEGENC_Obj {
    struct IJPEGENC_Fxns *fxns;
} IJPEGENC_Obj;

/*
 *  ======== IJPEGENC_Params ========
 *  This structure defines the creation parameters for all JPEGENC objects
 */
typedef struct IJPEGENC_Params {
    Int size; /* must be first field of all params structures */
    unsigned int  sample_prec;
    unsigned int  num_comps;
    unsigned int  num_qtables;
    unsigned int  interleaved;
    unsigned int  format;
    unsigned int  quality;
    unsigned int  num_lines[3];
    unsigned int  num_samples[3];
    unsigned int  output_size;

} IJPEGENC_Params;
typedef IJPEGENC_Params IJPEGENC_Status;
/*
 *  ======== IJPEGENC_PARAMS ========
 *  Default parameter values for JPEGENC instance objects
 */
extern IJPEGENC_Params IJPEGENC_PARAMS;
```

```
/*
 *  ======== IJPEGENC_Fxns ========
 *  This structure defines all of the operations on JPEGENC objects
 */
typedef struct IJPEGENC_Fxns {
    IALG_Fxns ialg;    /* IJPEGENC extends IALG */
    XDAS_Bool   (*control)(IJPEGENC_Handle handle, IJPEG_Cmd cmd, IJPEGENC_Status
*status);
    XDAS_Int32  (*encode)(IJPEGENC_Handle handle, XDAS_Int8* in, XDAS_Int8* out);

} IJPEGENC_Fxns;

#endif  /* IJPEGENC_ */
```

4.3 Encoder Performance

The application has been developed predominantly using C and serial assembly. . The DCT routine, `fdct_8x8()`, is hand optimized and is from the C62x ImageLIB [6], which provides a collection of high performance C-callable routines that can serve as key enablers for a wide range of image/video processing applications. Table 1 tabulates the frame rates at various resolutions that can be obtained with a 150 MHz C6211. However, it does **not** include routines for color-space conversions or format conversions. Further performance optimizations are possible. The performance may vary depending on the complexity of the images and the image formats like 4:2:0, 4:2:2, 4:4:4. Only 4:2:0 images are used for the encoder performance test. Those cycle counts reflects the DAT data transfer functions of the CSL (Chip Support Library) version 1.20 [5] instead of user-defined DMA functions. See section 4 for further JPEG Encoder performance data.

Table 1. JPEG Encoder Performance

Image Resolution			Performance		
	Width	**Height**	**Cycles**	**Clk/Blk**	**Frm/s**
128	128	128	420,997	1,096	356.30
256	256	256	1,504,753	980	99.68
CIF	352	288	2,307,620	971	65.00
VGA	640	480	6,605,576	917	22.71
SDTV	720	480	7,393,523	913	20.29

Note: All performance data is for 4:2:0 imagery.

5 Decoder

5.1 Description

The JPEG standard is a broad standard encompassing several compression and transmission modes. In order to facilitate future expansion to other modes, this implementation has a very modular construction. A single thread of control code handles all individual routines (kernels) which are called multiple times as required by the application. This control code will always be in ‘C’ to facilitate changes in control architecture.

The control function `jpgdec_ti()` chronologically calls all the decoder component routines like variable length decoding, run-level decoding, inverse quantization, IDCT etc., and performs data translations between the decoder routines. It operates the required double buffering scheme for DMA read-in of the JPEG bitstream from the external memory and DMA write-out of image data to the external memory. It takes care of parallelizing the data transfers with the core decoding kernels.

Figure 2 shows the block diagram for the control flow of the decoder in its present implementation.

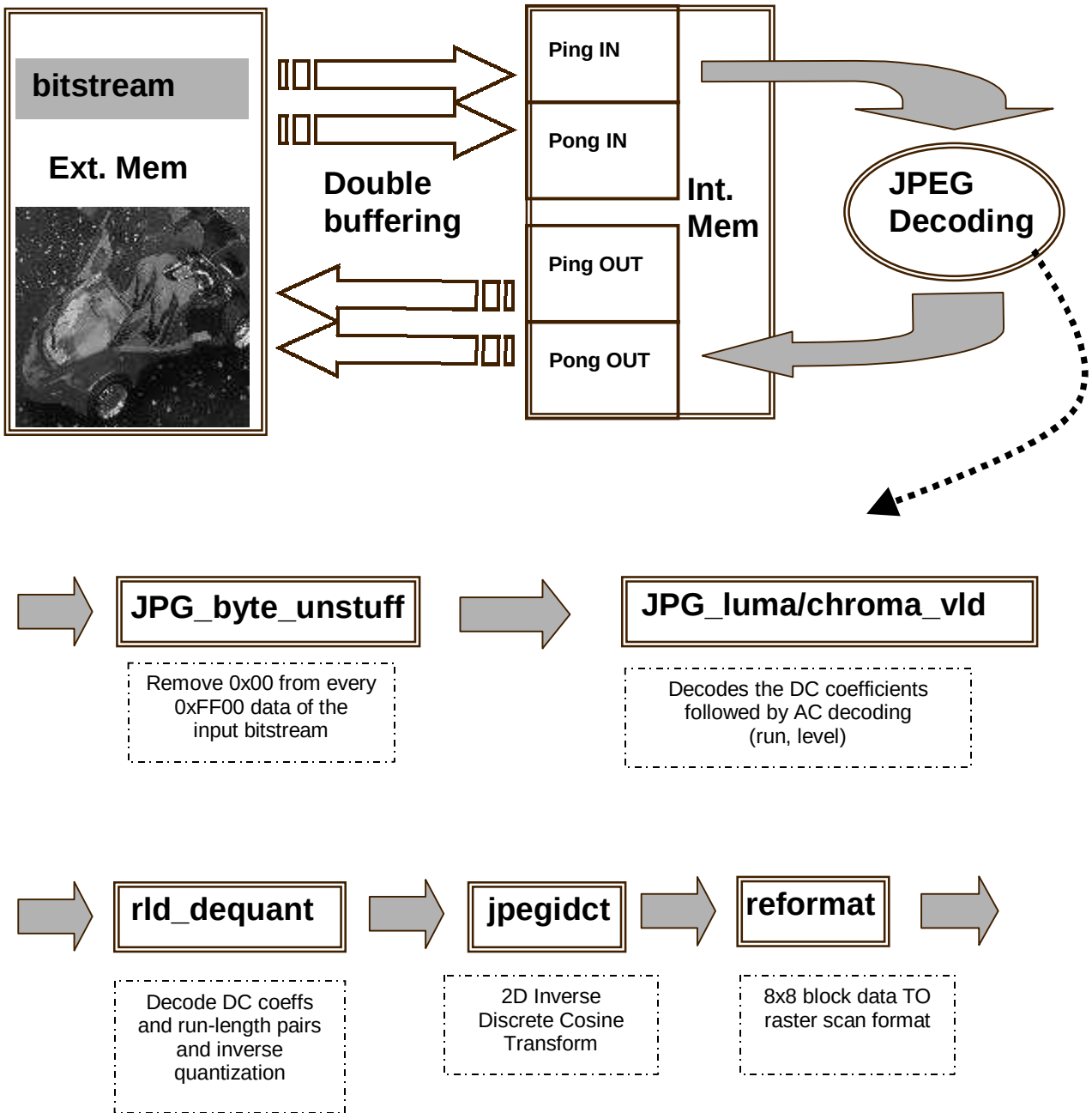

Figure 2. Block Diagram for the Control Flow of the Decoder

The decoder is capable of providing limited number of error codes. It returns a positive integer representing the number of output bytes when successful, or a negative integer when it encounters errors in the JPEG bit-stream.

Positive integer: Number of bytes of output data.

1. SOI not Found
2. Not Baseline
3. Quant-table not found
4. SOS not found
5. Interleaved, not supported

5.2 Decoder API

The API wrapper is derived from template material provided in the TMS320 DSP Algorithm Standard documentation. Knowledge of the algorithm standard is essential to understand the API wrapper. Please see the algorithm standard documentation for details on the TMS320 DSP Algorithm Standard. A complete discussion on how to make the algorithm eXpressDSP compliant is beyond the scope of this document, however the algorithm interface will be discussed as knowledge of this ensures inter-operability of algorithms. An algorithm is said to be eXpress DSP compliant if it implements the IALG Interface and observes all the programming rules in the algorithm standard. The core of the ALG interface is the IALG_Fxns structure type, in which a number of function pointers are defined. Each eXpressDSP-compliant algorithm **must** define and initialize a variable of type IALG_Fxns. In IALG_fxns, algAlloc(), algInit() and algFree() are required, while other functions are optional.

```
typedef struct IALG_Fxns {
    Void    *implementationId;
    Void    (*algActivate)(IALG_Handle);
    Int     (*algAlloc)(const IALG_Params *, struct IALG_Fxns **, IALG_MemRec *);
    Int     (*algControl)(IALG_Handle, IALG_Cmd, IALG_Status *);
    Void    (*algDeactivate)(IALG_Handle);
    Int     (*algFree)(IALG_Handle, IALG_MemRec *);
    Int     (*algInit)(IALG_Handle, const IALG_MemRec *, IALG_Handle, const
                       IALG_Params *);
    Void    (*algMoved)(IALG_Handle, const IALG_MemRec *, IALG_Handle, const
                        IALG_Params *);
    Int     (*algNumAlloc)(Void);
} IALG_Fxns;
```

The algorithm implements the algAlloc() function to inform the framework of its memory requirements by filling the memTab structure. It also informs the framework whether there is a parent object for this algorithm. Based on information it obtains by calling algAlloc(), the framework then allocates the requested memory. AlgInit() initializes the instance persistent memory requested in algAlloc(). After the framework has called algInit(), the instance of the algorithm pointed to by handle is ready to be used.

To delete an instance of the algorithm pointed to by handle, the framework needs to call algFree(). It is the algorithm's responsibility to set the addresses and the size of each memory block requested in algAlloc() such that the application can delete the instance object without creating memory leaks.

The API for the JPEG Decoder is:

```
/*
 *  ======== ijpegdec.h ========
 *  IJPEGDEC Interface Header
 */
#ifndef IJPEGDEC_
#define IJPEGDEC_

#include <xdas.h>
#include <ialg.h>
#include <ijpeg.h>



/*
 *  ======== IJPEGDEC_Handle ========
 *  This handle is used to reference all JPEG_DEC instance objects
 */

typedef struct IJPEGDEC_Obj *IJPEGDEC_Handle;

/*
 *  ======== IJPEGDEC_Obj ========
 *  This structure must be the first field of all JPEG_DEC instance objects
 */
typedef struct IJPEGDEC_Obj {
    struct IJPEGDEC_Fxns *fxns;
```

```
} IJPEGDEC_Obj;
/*
 *  ======== IJPEGDEC_Params ========
 *  This structure defines the creation parameters for all JPEG_DEC objects
 */
typedef struct IJPEGDEC_Params {
    Int size; /* must be first field of all params structures */
} IJPEGDEC_Params;
/*
 *  ======== IJPEGDEC_Status ========
*  This structure defines the status parameters for all JPEG_DEC objects
*/
typedef struct IJPEGDEC_Status {
    Int size; /* must be first field of all params structures */
    unsigned int    num_lines[3];
    unsigned int    num_samples[3];
    unsigned int    gray_FLAG;
    unsigned int    outputSize;
} IJPEGDEC_Status;
/*
 *  ======== IJPEGDEC_PARAMS ========
 *  Default parameter values for JPEG_DEC instance objects
 */
extern IJPEGDEC_Params IJPEGDEC_PARAMS;
/*
 *  ======== IJPEGDEC_Fxns ========
 *  This structure defines all of the operations on JPEG_DEC objects
 */
typedef struct IJPEGDEC_Fxns {
    IALG_Fxns ialg;    /* IJPEGDEC extends IALG */
    XDAS_Bool  (*control)(IJPEGDEC_Handle handle, IJPEG_Cmd cmd, IJPEGDEC_Status
*status);
    XDAS_Int32 (*decode)(IJPEGDEC_Handle handle, XDAS_Int8 *in, XDAS_Int8 *out);
} IJPEGDEC_Fxns;
#endif  /* IJPEGDEC_ */
```

5.3 Decoder Performance

The application has been developed predominantly using C and serial assembly. The functions `ac_vld_decode()` and `jpegidct()` are hand optimized. The `jpegidct()` is a slight modification from the one in ImageLIB [6]. The cycle counts do **not** include routines for color-space conversions or format conversions. Further performance optimizations are possible. Table 2 tabulates the frame rates at various resolutions that can be obtained with a 150 MHz C6211. However, it does **not** include routines for color-space conversions or format conversions. The performance may vary depending on the complexity of the images and the image formats like 4:2:0, 4:2:2, 4:4:4. Only 4:2:0 images are used for the decoder performance test. Those cycle counts reflects the DAT data transfer functions of the CSL (Chip Support Library) version 1.20 [5] instead of user-defined DMA functions. Please also see section 4 for further JPEG Decoder performance data.

Table 2. JPEG Decoder Performance

Image Resolution			Performance		
	Width	**Height**	**Cycles**	**Clk/Blk**	**Frm/s**
128	128	128	422,762	1,101	354.81
256	256	256	1,416,647	922	105.88
CIF	352	288	2,122,491	893	70.67
VGA	640	480	5,843,410	812	25.67
SDTV	720	480	6,537,918	807	22.94

Note: All performance data is for 4:2:0 imagery.

6 Optimization Techniques for the TMS320C6211

Cache performance is of primary importance in achieving overall best performance on theTMS320C6211, a 2-level cache high performance DSP. The TMS320C6211 has L1 data cache (L1D), L1 Instruction cache (L1I) and L2 unified data and instruction SRAM/Cache. Various methods are used to optimize each of the 3 separate caches above. Efficient use of the DMA is another important issue to exclude extra overhead, when double buffering scheme is used as in many DSP applications. The right compiler options can also make a significant difference to the end performance.

Following are the optimization techniques utilized in this JPEG implementation and they will be covered in the following sections in the order of importance.

- Choosing optimal sizes of the intermediate data buffers to fit into the L1 data cache.
- Configuring the L2 Cache/SRAM ratio.
- Code aligning to minimize the L1 instruction cache thrashing.
- Efficient use of DMAs.
- Choosing performance-critical compiler options.

- General optimization techniques for the TMS320C6000 DSP family.

There is possibility of L2 cache corruption because of the lack of coherency between L1/L2 and the external memory. The last section details all the possible cases of corruption. The solution for each case is described.

6.1 Basic Understanding of the TMS320C6211 2-level Cache Architecture

The TMS320C6211 DSP has two level cache, the L1 and the L2, while the TMS320C6201 has flat memory structure that consists of 64KB data memory and 64KB program memory. The L1 cache is divided into 4KB L1I (Instruction cache) and 4KB L1D (Data cache).

The L1I always operates as a direct-mapped cache, and may not be memory mapped. Its line size is 64 Bytes that correspond to 2 fetch packets in the TMS320C6000 family, which will provide an amount of program pre-fetch on a miss to the L2/external memory. The throughput of the L1I is a single cycle.

The L1D is a 2-way set associative data cache and accessed in a single cycle by the CPU and its line size is 32 Bytes. This cache may not be disabled to operate as a memory mapped SRAM. The L1D shall cache-access to all the L2 space and any external addresses identified as cacheable by MAR registers. The L1D controller implements Least Recently Used (LRU) algorithm to determine which line is to be replaced.

The L2, the second level of internal memory, is a unified 64KB data and instruction RAM/Cache. The user may select portions of the L2, in ¼ L2 size increments, to be configured as SRAM for either program or data segments. It operates maximum 4-ways in the full 64KB cache mode. 0 illustrates this associativity.

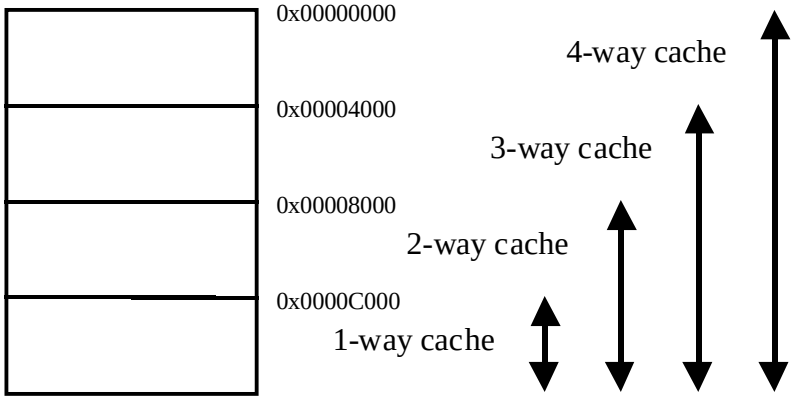

Figure 3. L2 Cache and SRAM Configurations on C6211

6.2 Memory-mapped L2 Control Registers for Cache Operations

The L2 controller provides several memory-mapped control registers and the CPU can access those registers to configure the L2 operating mode. In order to eliminate a large number of potential lockup conditions in the system, the L2 control registers are accessible as read-only registers to I/O subsystem. Only the CPU can write-access the registers. Following are the L2 control registers:

L2CFG: controls the spilt between cache and SRAM. The three Least Significant Bits (L2MODE) determine the combination of cache and SRAM.

L2 Mode	Cache / SRAM Model
000 (reset value)	64K SRAM, D/I mixed
001	16K 1-way D/I cache, 48K SRAM
010	32K 1-way D/I cache, 32K SRAM
011	48K 1-way D/I cache, 16K SRAM
100 – 110	Reserved
111	64K 1-way D/I cache, 0K SRAM

WBAR / WWC:	Write Back
WIBAR / WIWC:	Write Back with Invalidate
IIBAR / IIWC:	Invalidate instruction
FDBAR / FDWC:	Flush data
L2CLEAN:	Clean L2 cache space (not including L2 SRAM)
L2FLUSH:	Flush L2 cache space (not including L2 SRAM)
MARn:	Define the cache-ability of the external memory spaces

For more details, see the TMS320C6211 Product Specification revision 1.31[4].

It is not required to directly set those L2 control registers any more, by introducing the Chip Support Library (CSL). The CSL is a set of application programming interfaces (APIs) used to configure all on-chip peripherals for all the TMS320C6000 devices. Appendix A shows the corresponding CSL API functions to manage the L2 program/data cache and the SRAM.

6.3 Optimal Data Size for L1 Data Cache Operation

It is important to limit the total amount of internal data buffers to exactly fit into the L1D data cache for best performance. The L1D is a 2-way set associative data cache of total 4 KB. The L1D controller implements Least Recently Used (LRU) algorithm to determine which line is to be replaced.

In many data-oriented applications, a series of functions works like a conveyor belt system. A data buffer is passed to a function as an input and then this function usually outputs a data buffer. The output data buffer is passed to the next function as an input data buffer and so on. The current version of JPEG works virtually in the same way except for some conditional branches.

One-way of the 2 KB L1 data cache shall be used for caching the input buffer and the other 2KB for caching the output buffer on the L2 SRAM. Data buffers bigger than 2KB will degrade the performance by increasing the L1D cache thrashing.

JPEG requires an 8x8 block as the minimum amount of data unit for processing, which corresponds to 64 bytes in 8-bit precision. As mentioned above, each way of the L1D is 2KB and every input and output data buffer for every kernel must not be bigger than 2KB to avoid unnecessary L1D cache thrashing. Maximum number of 8x8 blocks for each input and output are 32 (32x8x8 = 2048 Bytes) for 8-bit precision, but some of the JPEG kernels (DCT/IDCT) require that every 8-bit pixel expand to 16-bit precision. A total 16 of 8x8 blocks will thus completely fill one way of the L1D cache
(16x8x8x2 = 2048 Bytes).

16 8x8 blocks corresponding to 2 KB (16x8x8x2 = 2048 Bytes), assuming the precision is 8 bit, will virtually guarantee no cache thrashing, since it keeps input/output data buffers to or within 2 KB space of each L1D way for all the kernels. The current C6211 JPEG keeps all the buffers to exactly 2 KB while the C6201 JPEG processes as many blocks as possible at a time to reduce the overhead ratio of the control logic. Larger number of blocks per iteration always guarantees better performance on the C6201, depending on the internal memory availability.This may not be true, however, on the 2-level cache TMS320C6211DSP.

Making the data buffers either much smaller or bigger than 2 KB will degrade the performance. Data buffers bigger than 2 KB cause more cache thrashing (cache miss). Data buffers smaller than 2 KB may not introduce cache thrashing, but result in less cache efficiency. Smaller buffers cause extra overhead to the main flow control routine by increasing the number of iterations

Table 3 shows the overall performance variation of the JPEG decoder for different levels of L1 data cache utilization. The same image of 128x128 resolution was used for all the different setting simulations. The cycle counts are measured on the C6211 DSK. The number of blocks means that N number of 8x8 blocks are processed during a single iteration of the major loop in the JPEG decoder. As shown, 16 blocks per processing resulted the best performance regardless of the L2 cache/SRAM configuration. The L1D has major impact on the C6211 JPEG encoder and decoder performance. If the number of blocks is greater than 16, more internal buffer space is required, which causes the L2 SRAM to exceed 16KB. That is why 24/32 blocks data is based on 32KB cache space.

Table 3. Performance Analysis of the L1D Cache Efficiency

Num of Blocks	L2			Decoder			Performance (%)
	Cache/SRAM	W	H	cycles	Clk/Blk	Frm/s	
1	48/16 KB	128	128	841,766	2,192	178.20	50.2
2	48/16 KB	128	128	613,409	1,597	244.54	68.9
4	48/16 KB	128	128	495,977	1,292	302.43	85.2
8	48/16 KB	128	128	438,611	1,142	341.99	96.3
16	48/16 KB	128	128	422,471	1,100	355.05	100.0
16	32/32 KB	128	128	422,328	1,100	355.17	100.0
24	32/32 KB	128	128	430,864	1,122	348.14	98.1
32	32/32 KB	128	128	436,318	1,136	343.79	96.8

Processing only one block per iteration requires 16 times more looping than processing 16 blocks per iteration. This causes heavy looping overhead and a big portion of the L1D memory space is not fully utilized, giving only half (50.2%) of the overall performance. Those applications with very tight internal memory requirements should consider the trade-off between data size and performance as shown in Table 3.

However, choosing internal buffers bigger than one way of L1 data cache size, 2KB is not recommended. It causes performance degradation as well as occupying more L2 space. Table 3 shows almost no difference between a 32KB cache and a 48KB cache for 16 blocks per processing. The slight difference is well within error range (0.03%). The same performance is because the single tasking baseline JPEG is a relatively small application. The total code size of the JPEG decoder is less than 26KB and is can fit within either 32KB or 48KB of the L2 cache. Large applications may result in more performance difference according to the various L2 cache/SRAM configurations.

The current JPEG encoder utilized the same techniques used in the decoder.

6.4 Performance Issues on L2 Cache/SRAM Configuration

The C6201 JPEG implementation requires 41 KB of data memory for the encoder and 36 KB for the decoder, which are within the 64KB limit of data memory, to maximize the performance. However, on the C6211, more data memory reduces the L2 cache space and less cache space reduces the associativity, which degrades the cache performance, hence the overall performance degrades. Therefore, it is important for many C6211 applications to reduce the SRAM requirements to less than 16 KB. That corresponds to one cache block of the L2 memory. The SRAM requirement of 16 KB or less gives 48 KB of L2 cache space and 3-way associativity. The associativity is as important as the amount of the cache space itself, because more associativity means that the data or instruction can be cached from more locations of the next level memory (external memory for the L2 cache) simultaneously. For example, 2-way associativity cache has to thrash the Lease Recently Used (LRU) cached-data when the third data is cached, while 3-way associativity cache can still cache the third data without thrashing any of the already cached data.

While the internal data and program memory requirement should be minimized from the view of the L2 memory, the data memory requirement should be matched with the maximum utilization of the L1 data cache. The major buffers in the internal memory should fit into the 4 KB, 2-way L1 data cache space to prevent undesired cache thrashing during repetitive kernel calls inside the main loop.

Table 4 shows the performance variations according to the various L2 cache configurations for several image resolutions. Assuming the performance of 48 KB cache / 16 KB SRAM as 100 %, the average performance at 16 KB cache / 48 KB SRAM is less than 90 % for the encoder and less than 95% for the decoder.

Table 4. Performance Analysis on Various L2 Cache Configurations

Image			Performance (%)					
Resolution			Encoder (Cache/SRAM)			Decoder (Cache/SRAM)		
	W	H	48/16 KB	32/32 KB	16/48 KB	48/16 KB	32/32 KB	16/48 KB
256	256	256	100	100	100	100	100	93
CIF	352	288	100	100	86	100	100	94
VGA	640	480	100	97	89	100	100	94
SDTV	720	480	100	97	90	100	100	94

To fit the requirement of the 16 KB L2 SRAM for the internal memory, the current C6211 JPEG code was constructed such that the encoder requires 11 KB of internal SRAM space and the decoder requires 14 KB. However, the actual usage of SRAM space is 16 KB for both the encoder and decoder, since the L2 memory is divided into a multiple of 16 KB and only 48KB of memory is used as L2 cache even though we use less SRAM than 16 KB. The 16 KB SRAM space mainly includes the stack space plus additional blocks like .bss, .const or user-defined table sections in the current implementation of the TI JPEG code.

6.5 Code Alignment for the L1 Instruction Cache Operation

The L1I is a one-way 4KB instruction cache on the L1 level of the memory hierarchy. Unlike data caches which need at least 2-way associativity, instructions usually do not have to be written back to the original memory location after processing and are very sequential. That is why multi-way is not a critical issue for instruction caches.

Because the L1I cache is direct mapped, each address maps to a unique location within the cache. The replacement scheme is straightforward and implemented by masking off the upper 20 bits of the requested address and mapping the lower 12 bits to the physical RAM address of the cache. L1I address is organized as tag, set and offset. Tag is the upper 20 bits, set is the next 6 bits, and offset is the lower 6 bits.

There will be virtually no instruction cache miss inside the major loop which takes most of the total cycles, if all the critical kernels including corresponding main control code within the major loop can fit into the 4 KB L1I cache.

The TI JPEG code is too large an application to fit into the 4KB L1I cache, so we could not take the desired advantages by aligning instructions in an L1I friendly manner. For example, the code sizes of the critical kernels inside the major loop of the JPEG decoder are:

JPG_luma_vld()	3A0h bytes
JPG_chroma_vld()	3A0h bytes
ac_vld_decode()	340h bytes
rld_dequant()	180h bytes
jpegidct()	540h bytes
reformat()	160h bytes
Total	12A0h bytes (4.66 KB)

The code size of all the kernels, 4.66 KB, plus the corresponding code size of the control logic inside the major loop exceeded the 4KB capacity of the L1I. In addition, `DAT_CopyXX()` of the CSL is included to transfer the processed data to the external memory. It occupies 0.5 ~ 1 KB of memory. Therefore, at least a few kernels are expected to be partially thrashed from the L1I cache during every iteration.

Our experiments showed that intentionally aligned codes slightly increased the total cycles counts, less than 0.5%, compared to the cycle counts without intentional code alignment. This is because non-aligned codes have less overlapped instructions in between the critical kernels above.

Another experiment resulted in slight improvement. Since `JPG_luma_vld()` is used only for luminance processing and `JPG_chroma_vld()` only for chrominance processing and they are guaranteed not to be used in combination in a single iteration, we aligned the other kernel codes sequentially and placed `JPG_luma_vld()` and `JPG_chroma_vld()`exactly 4KB apart from each other, which resulted in those two kernels overlapping in the 4 KB L1I range.

Figure 4 illustrates the code alignment explained above. Kernels A1 and A2 in the figure indicate `JPG_luma_vld()`and `JPG_chroma_vld()`. Other kernels are illustrated as B, C and D. The processing sequence is A1-B-C-D for luminance processing and A2-B-C-D for chrominance processing. Before the alignment, kernels A1 and D are thrashed during every iteration of the luminance processing, since they occupy the same portion of the L1 instruction cache. The kernels A2 and D are not thrashed for chrominance processing.

After the code alignment, either order of iteration, A1-B-C-D or A2-B-C-D, does not affect the efficiency of the L1I. This technique improved the performance by 0.5% for 128x128 image and 0.7% for CIF (352x288) image. Part of the control logic code can be cached in-between the kernels, causing the overlapping and thrashing of the kernels, but both of the alignment methods in Figure 4 have the same possibility of the overlapping. Thus, the alignment technique is still applicable, regardless of the existence of control logic.

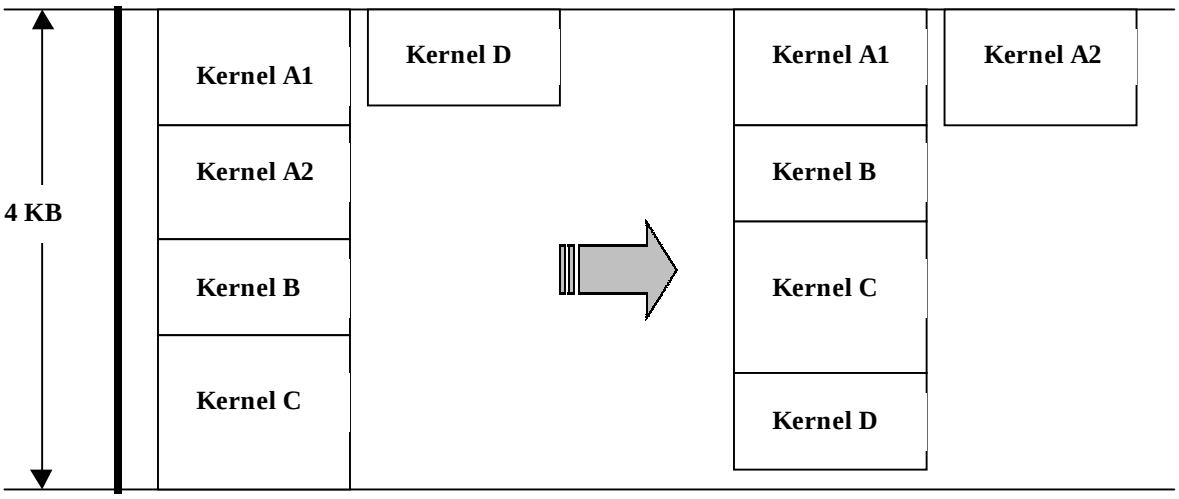

Figure 4. Code Alignment by Locating Two Kernels 4 KB Apart from Each Other

The improvement of 0.5% may not be noticeable. However, considering the alignment was done only on some of the kernels, we may get better results when the code alignment is applicable to all of the critical kernels. It is advantageous that this improvement can be obtained simply by aligning kernel codes to fit optimally within the L1 cache, without any coding effort.

Even though the current JPEG is not a good candidate for the code aligning technique, it may be useful to apply the idea to other C6211 applications, since it may boost the performance without any coding effort or extra complication.

6.6 Efficient DMAs Maximizing Double Buffering Scheme

Many DSP applications require Direct Memory Access (DMA) in the background, since DSPs usually have limited amount of internal memory for both program and data, while enough external memory space can be supported as needed. The TMS320C6211 has 64 KB of user-configurable internal memory space. Even with this limited space, the JPEG encoder and decoder uses only 16KB of the internal L2 memory each as SRAM because of the performance issues related with L1 and L2 cache operation as mentioned earlier.

The TMS320C6000 family of VLIW DSPs has a separate DMA controller which controls data transfers without imposing extra burden on the CPU. The TMS320C6201 has efficient DMA functions with 4 channels. A new mechanism, EDMA, was added to TMS320C6211, to enhance the existing C6201 DMA, including link capabilities and up to 16 channels.

The QDMA, or quick DMA, controller supports the same transfer modes as the EDMA mechanism, however, it is quicker to submit. EDMA may be used for periodic real-time peripheral servicing, but QDMA is very suitable for the JPEG implementation, since all the data are transferred in blocks.

The QDMA controller supports various dimensionalities, 1D, 1D-to-2D, 2D-to-1D and 2D-to-2D transfers. However, because the QDMA does not support 2D-to-2D DMA with separate pitches for source and destination, it is important to reorganize, if needed, the buffers not to require such DMAs. It will introduce possibly severe extra overhead to trigger 8 times of 1D DMAs as an alternative of the 2D-to-2D DMA with separate pitches or trigger any type of DMA that is not supported by the QDMA controller. The current JPEG utilizes the 1D DMA for the encoder output and the decoder input to transfer the JPEG bitstreams. Also used are 2D-to-1D and 1D-to-2D DMAs for the encoder input and the decoder output, where 2D means raster scan format and 1D means reformatted form as a series of 8x8 blocks data for more efficient processing. A double buffering scheme is used in the same way as in the JPEG implemented on TMS320C6201 [3].

6.7 Performance-Critical Compiler Options

There are several performance-sensitive compiler options. First, you may want to choose the maximum optimization level of compilation by setting ‘-o3’. ‘-g’ option is used for debugging and testing purpose and adds 5~10 % overhead, so ‘-g’ or ‘-mg’ must be removed for the final performance.

Another critical option is ‘-mt’, which assumes that certain aliasing techniques are not used. This may boost up to 20~30 % of the overall performance depending on the applications. Table 5 shows the impact of the ‘-mt’ option in the total cycles of the JPEG decoder with a 128x128 image. The JPEG decoder achieved 13.4 % performance improvement by simply adding the option ‘-mt’.

The JPEG decoder includes 2 hand-coded assembly kernels that consume 47 % of the overall CPU cycles. Those hand-coded kernels are not affected by the compiler options. The actual improvement for the ‘C’ kernels, therefore, is not 13.4 %, but 25.3 % (=13.4/53.0). The ‘C’ portion of the TMS320C6000 applications improve up to 30 % depending on the applications.

The option ‘-mt’ must be used extremely carefully because of the restrictive assumptions on the aliasing of pointers. Those assumptions are described in detail in [7]. If any of the aliasing techniques in [6] is used, ‘-ma’ must be used instead of ‘-mt’, which will add extra overhead. The JPEG encoder and decoder are free from those assumptions, giving the best performance with the ‘-mt’ compiler option.

Table 5. The Impact of ‘-mt’ Option on the JPEG Decoder Performance

Compiler options	W	H	Cycles	Clk/Blk	Frm/s	Performance (%)
‘-o3 –ml0 –mt’	128	128	420,209	1,094	356.97	113.4
‘-o3 –ml0 -ma’	128	128	485,200	1,264	309.15	98.2
‘-o3 –ml0’	128	128	476,337	1,240	314.90	100.0

6.8 General Optimization Techniques for the TMS320C6000 DSP Family

Double buffering frees the CPU from the heavy load of data transfers in many image and video processing applications, utilizing the capability of the DMA or EDMA controller provided by the TMS320C6000. The idea is to parallelize the CPU processing and the data transfer by the DMA or EDMA controller with two same-sized buffers. On the input side of the DMA, one buffer is used as an active buffer for data processing and the other is used as a passive buffer for data transfer until the active buffer is emptied by the processing and the passive buffer is filled by the DMA-in transfer from external memory. The two buffers are switched for the next run, and so on. On the output side of the DMA, the active buffer is filled with the processed data by the CPU and the passive buffer is emptied by the DMA-out transfer. The JPEG encoder and decoder adopt double buffering at both ends of the application, input and output. All the data transfers are on background except the pre-fetch of the first input DMA packet and the post-delivery of the last output DMA packet.

The TMS320C6211 has 64 KB of internal memory space and it is recommended to use as many L2 cache blocks as possible as explained in section 6.4. Since programs are usually sequential and cached very well via the L1 instruction cache and the L2 cache, the code may be placed in the external memory.

The buffers for double buffering scheme must be placed in the internal memory whether they are in the stack or heap, because external to external DMAs are not only very slow but also undesirable. In addition, there is a critical issue on the external coherency with the internal memories, which is described in the following section. Other sections can be placed in the external memory, but it is highly recommended to put the frequent user-defined tables in the internal memory as long as the memory requirement allows, preferably within one block of 16KB L2 memory.

6.9 Avoiding Undesired L2 Cache Corruption

Due to the multi-level memory hierarchy and the lack of coherency between the L2 memory and the external memory on the TMS320C6211, it is not recommended to access external memory locations directly and by DMA at the same time. This may cause CPU or DMA controller to access old data in the external memory while new data is still in cache space.

6.10 Coherency

Coherency is maintained by tracking the state of lines in the L1D and L2. The coherency between the data and instructions side was removed to simplify the architecture, which removes significant hardware complexity and speed paths, at the cost of hardware supported self-modifying code. Another hardware simplification is that external memory coherency is not maintained with the L1D and L2. Since many applications including the JPEG implementation does not cache instructions into the L1 data cache, the coherency between L1I and L1D is not a usual problem for many applications. Self-modifying codes will be required for the rare applications that cache instructions into the L1D. However, external coherency may affect applications like JPEG.

There are three types of possible cache corruption caused by lack of the external coherency with the internal L1/L2 memory. The DMA may read old data when the CPU writes to an external memory location and the DMA reads the data, since the data written by the CPU stays in the L1/L2 space instead of being actually written to the external memory location. The CPU may read old data when the DMA writes first and then the CPU reads. Correctness is not guaranteed when the CPU and the DMA write to the same external location, since the CPU write may not be directly written to the memory location. Following examples illustrate these three cases in detail.

6.10.1 CPU Writes and DMA Reads Old Data

Following is a simple example code illustrating how the lack of external coherency can result in corrupted data for the DMA data transfer.

```
 Val = *val_adrs;  /* *val_adrs = 10 */
*val_adrs = val + 10; /* *val_adrs = 20 */
DMA from val_adrs to destination; /* transfer 20 to dest adrs */
```

The DMA in the code above must transfer the value ‘20’, it is not intending to transfer the value ‘10’. However, it is possible for the DMA to transfer ‘10’ if the pointer `val_adrs` is located in the external memory where coherency is not maintained with the internal memory.

Figure 5 shows how the lack of external coherency can affect the correctness of the data written by the CPU.

1. The CPU reads data from an external memory location. The data is cached into the L2 memory and then cached into the L1 data cache. Now the same data resides in 3 locations, the original location, the L2 cache and the L1 data cache. Coherency is maintained between the L1D and the L2.
2. CPU processes the read data and writes to the original location. However, due to the cache, the data is written to the L1D or L2 cache space and stays there, not directly going to the original location.
3. DMA reads the original external location and the data is not updated yet due to the lack of the external coherency.

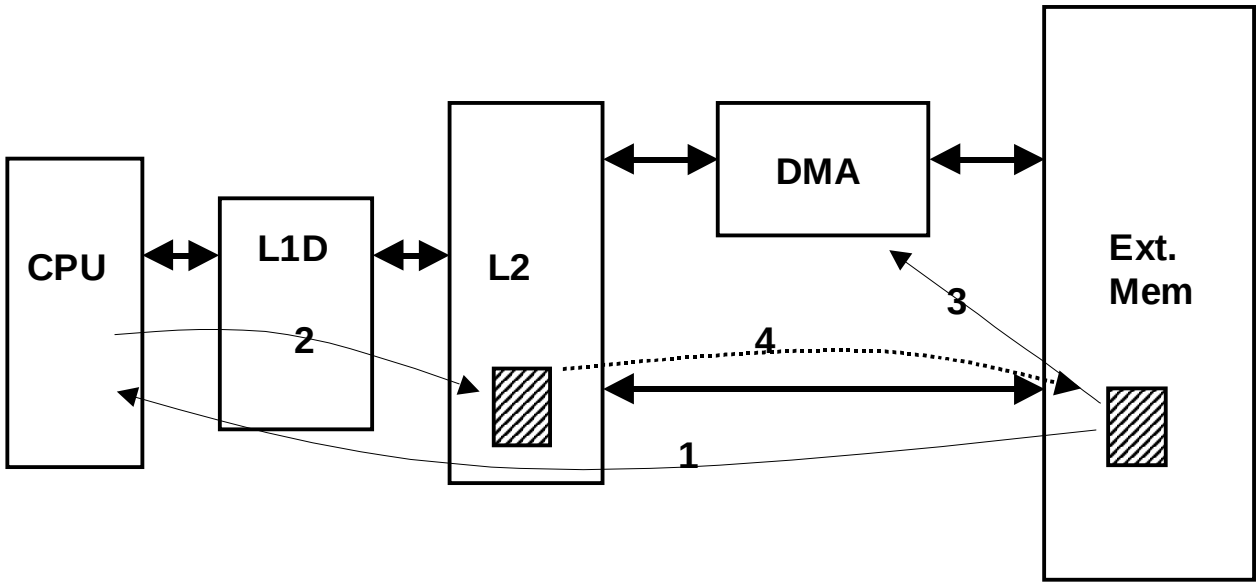

Figure 5. Example of 'CPU Write and DMA Read'

To maintain the external coherency, the write back from the L2 location to the external location should be done before the DMA reads the data. It will be guaranteed that the DMA reads correct new data if the operation is in the order of '1-2-4-3'. The TMS320C6211 provides simple ways to maintain the coherency by software. It maintains the coherency by write-back with invalidation to the external location, cleaning the L2 cache (L2CLEAN), or flushing the L2 cache (L2FLUSH) before the DMA accesses the external location to read. A set of memory-mapped registers is provided to do these operations. Details are described in [4] and [5]. Below is the correct code including the write back operation.

```
Val = *val_adrs;  /* val = 10 */
*val_adrs = val + 10; /* val = 20 */
L2CLEAN = 0x01;   OR    CACHE_Clean(CACHE_L2, start_adrs, end_adrs);
DMA from val_adrs to destination; /* transfer 20 to destination address */
```

The first method is directly setting the memory-mapped register, `L2CLEAN`, to trigger cleaning operation for the L2 cache [4] and the second method is using the `CACHE_Clean()` function provided by the Chip Support Library (CSL) [5].

This type of cache corruption happened in the early stage of the JPEG encoder implementation. The encoder output, i.e. the JPEG bitstream, was partially corrupted containing a few horizontal lines in the output image. The lines usually corresponded to 64 pixels (bytes) and the locations of the lines were almost random, but appeared to have a dependency on the previously processed image.

Below is a part of the old driver file for the JPEG encoder.

```
int main()
{
    /*=== read Y/Cb/Cr from 3 separate input files ===*/
    infile = fopen("input_file", "rb");
    fread (external_input_adrs, sizeof(unsigned char), width*height, infile);
    fclose(infile);

    /*=== Set Init parameter values in JPG_params [STRUCT] ===*/
    jpg_params.var1 = 1;    jpg_params.var2 = 2; ......

    /*=== Call the JPEG Encoder() ===*/
    ret_val = jpgenc_ti (jpg_params, external_input_adrs, external_output_adrs );

    /*=== Save the JPEG bitstream in output file ===*/
    outfile = fopen("output_file.jpg", "wb");
    ret_val = fwrite(external_output_adrs, sizeof(char), ret_val, outfile);
    fclose(outfile);
}
```

There was no coherency maintained between the L2 cache and the external memory, since the encoder processing started right after the `fread()` operation. The `fread()` was performed by the CPU and the CPU utilized the internal cache space, the L1D and the L2 cache. Thus, there were a few lines left in the cache even when the `fread()` operation was finished. Some of the lines in the external memory still had the previous image data when running multiple images, or bogus data when starting from rebooting, since the new data was not updated from the internal cache memory yet. The DMA of the JPEG encoder transferred the image data from the external memory to the internal data buffer, while some of the lines were still bogus or previous data. This is why the corruption depended on the previously processed images.

Flushing the L2 cache between the `fread()` operation and the encoder processing completely prevented the corruption.

6.10.2 DMA Writes, CPU Reads Old Data

0 shows how the DMA reads old data due to lack of external coherency.

1. The CPU reads data from an external memory location. The data is cached into the L2 memory and then cached into the L1 data cache. Now the same data resides in 3 locations, the original location, the L2 cache and the L1 data cache. Coherency is maintained between the L1D and the L2.
2. The DMA writes new data to the original external location and the data stays only in external memory, not being updated to the internal cache space.

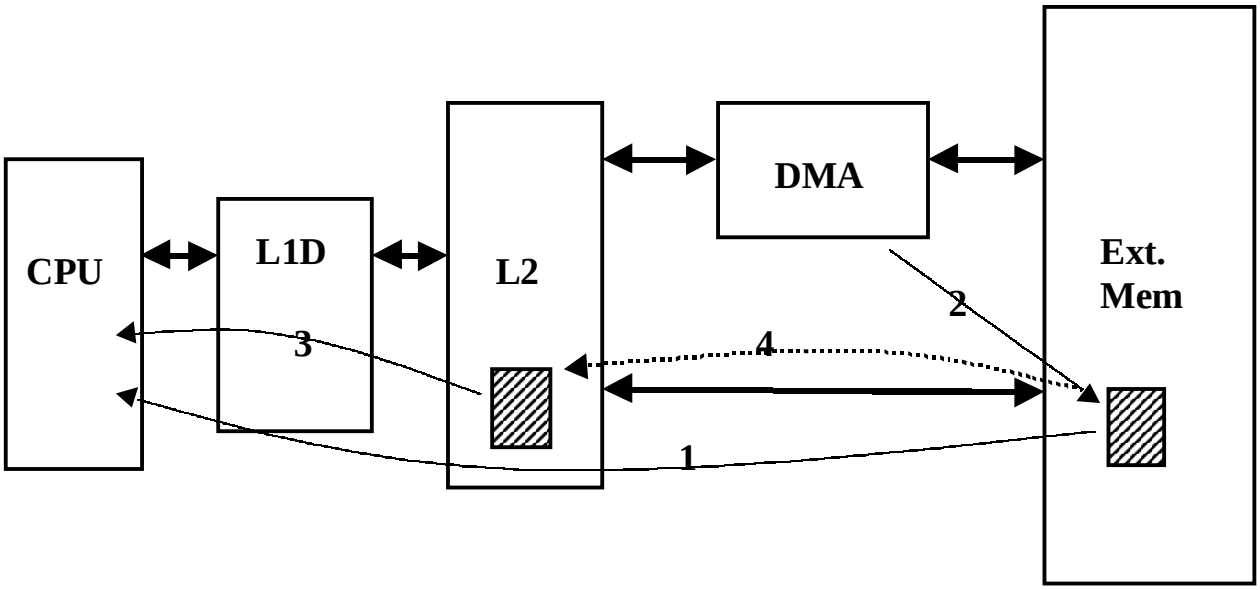

Figure 6. Example of 'DMA Write and CPU Read'

3. Now the CPU tries to read access the external location. The CPU looks for the data in the cache first, the L1D or the L2 and finds the data is still valid and not corrupted, since the DMA updated only the external location. The CPU reads the old data from the L2, not the new data from the external location.

To maintain the coherency, the old data in the cache space must be invalidated right after the DMA writes to the external location. The CPU accesses the new data for the next CPU read, once the old data is invalidated, since the CPU cannot find the data until it reaches the external location. The CPU brings the new data to the L2 cache and then to the L1D cache and finally to the CPU. The order of '1-2-4-3' is the correct operation and always guarantees that the CPU reads the new data. Invalidation is the only solution for this case, since other L2 operations like L2CLEAN/L2FLUSH write the cached data back to the external locations. There are two ways for the invalidation again, setting memory-mapped registers [4] and using CSL [5].

```
IIBAR = start_address; IIWC = number of words to invalidate; /* direct setting */
OR
CACHE_Invalidate(CACHE_L1DALL, 0x00, 0x00);  /* Chip Support Library */
```

6.10.3 Reordered CPU-DMA Writes

This case is less common than the former two cases, since the DMA or peripherals usually do not have to overwrite the data written by the CPU. However, this may happen when the CPU initializes a range of the external memory space and then the DMA writes to the range. Figure 7 shows how the reordering can occur.

1. The CPU initializes a range of the external memory, i.e. writes an initialization value to the range. The initialization data stays in the L2 cache instead of going to the external memory for some cache lines.

2. The DMA writes new data to the range of address.
3. The Initialization data is evicted to the external memory after the DMA write, which causes the reordering of the writes. The initialization data overwrites the new data from the DMA. The CPU will get the initialization value instead of the new data for the next read-access.

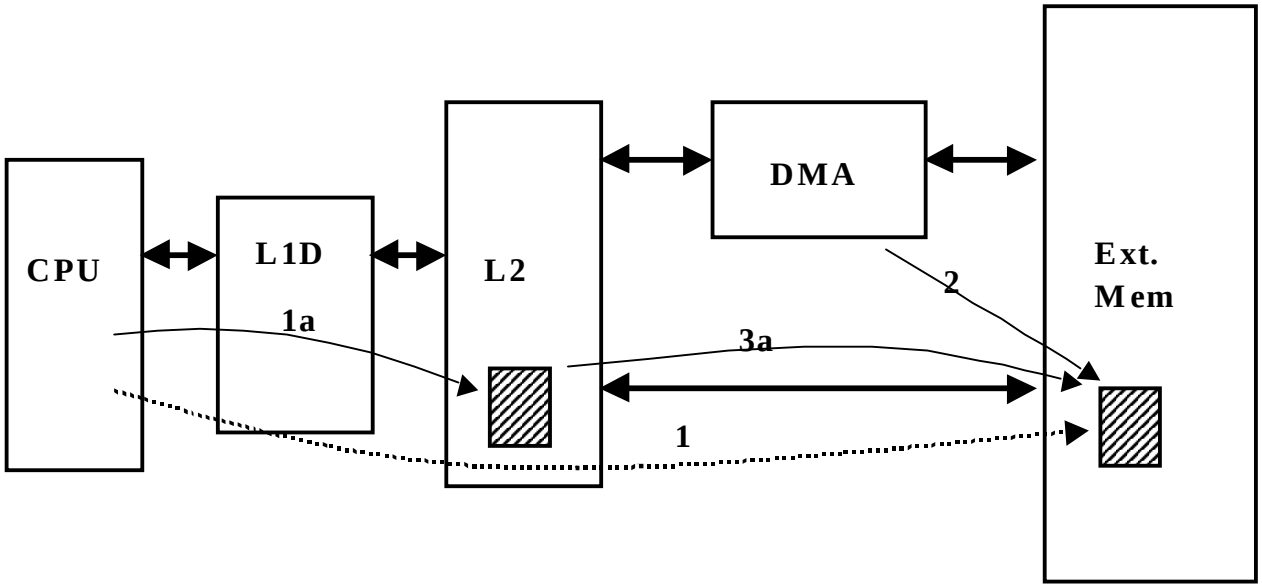

Figure 7. Example of 'Reordered Writes'

To maintain external coherency, the initialization must be completed before the DMA accesses the range of memory for write-transfer. All initialization data should be written to the external memory before the DMA. It is the write-back operation to guarantee all the dirty cache lines in the L1D/L2 are evicted to the external locations.

For all the three scenarios on cache corruption, the DMA represents any peripherals, which access the data written by the CPU or provide data for the CPU. The coherency problem is between the CPU and any peripheral, not just between the CPU and the DMA.

Different L2 configurations do not affect these scenarios, since the external coherency is not maintained either with the L1D or the L2 space. Even when the L2 is used in SRAM only mode, this coherency problem occurs between the L1D and the external memory. Cache clean and cache flush can be used as alternatives of the write back for the first and the third scenarios, but invalidation is the only solution for the second scenario.

7 Results

We have reviewed all possible optimization techniques on the TMS320C6211 except those techniques covered in the C6201 JPEG application note [3]. As a result of the optimization effort, Table 6 shows comparable performance to the C6211 JPEG encoder/decoder implementations when using the user-designed DMA routines.

Table 6. The C6211 JPEG Performance Over the C6201 JPEG Without CSL

Img Res.			ENCODER			vs. 6201 (%)	DECODER			vs. 6201 (%)
	W	H	Cycles	Clk/Blk	Frm/s		cycles	Clk/Blk	Frm/s	
128	128	128	392,885	1,023	382	89.5	400,647	1,043	374	94.5
256	256	256	1,416,884	922	106	90.6	1,384,283	901	108	90.7
CIF	352	288	2,176,496	916	69	88.5	2,077,660	874	72	89.7
VGA	640	480	6,289,602	874	24	89.5	5,773,518	802	26	89.2
SDTV	720	480	7,056,311	871	21	89.1	6,466,665	798	23	88.9

Note: All performance data is for 4:2:0 imagery.

Table 7 shows the performance when using the CSL-DMA routines, `DAT_CopyXX()`. The addition of the CSL DMA (DAT) to the C6211 JPEG implementation introduced 2~4 % of extra overhead depending on the image resolutions. The reasons are:

- The DAT DMA routines provided by the CSL are not as efficient as the user-defined DMA routines. Since the DAT modules had to take care of all the c62x family, they have bigger code sizes.

- The overhead of adding CSL to C6201 JPEG was within 1% while that of the C6211 JPEG was 4~7 %. This is because the old user-defined DMA routines on the C6201 were written in ‘C’ and not efficient.

- The addition of CSL slightly reduced the advantage of code alignment for the L1 instruction cache.

Table 7. The C6211 JPEG Performance Over the C6201 JPEG with CSL

Img Res.			ENCODER				DECODER			
	W	H	Cycles	Clk/Blk	Frm/s	vs. 6201 (%)	cycles	Clk/Blk	Frm/s	vs. 6201 (%)
128	128	128	420,997	1,096	356	83.5	422,762	1,101	355	88.0
256	256	256	1,504,753	980	100	86.2	1,416,647	922	106	87.7
CIF	352	288	1,504,753	971	65	84.3	2,122,491	893	71	87.1
VGA	640	480	6,605,576	917	23	86.0	5,843,410	812	26	87.5
SDTV	720	480	7,393,523	913	20	85.8	6,537,918	807	23	87.5

Note: All performance data is for 4:2:0 imagery.

8 Conclusion

All the optimization techniques have been reviewed to achieve the best performance on the TMS320C6211, the high performance VLIW DSP with dual level cache. It has internal memory of 4 KB each for L1D and L1I and 64 KB of unified L2 cache. The primary factor for the best performance was the L1D optimization as well as optimization of L2 and L1I performance. Other optimization techniques also have been reviewed including efficient use of DMAs and critical compiler options, and a few more common optimization techniques on the TMS320C6000 family.

After all these techniques are integrated, the JPEG implemented on TMS320C6211 performed up to 90 % without CSL and up to 88 % with CSL, compared to the performance of the TMS320C6201, the high performance DSP with 64KB each of data and program memory. As mentioned earlier, because of the L1D cache size, the performance of the C6211 JPEG can vary depending on the image resolutions. Images with width dimension that is a multiple of 128 usually resulted in better performance than others, since both the encoder and the decoder have limited internal buffers of size 2KB corresponding 128 x 8 pixels. The overall performance achieved before adding Chip Support Library was around 89 ~ 90 % of the C6201 JPEG for the common image formats like CIF, VGA and SDTV.

The optimization techniques introduced in this note can be applied to virtually any application on the TMS320C6211.

9 References

1. International Standard DIS 10918-1 CCITT Recommendation T.81, Digital Compression and Coding of Continuous-tone Still Images
2. JPEG STANDARDS by William B. Pennebaker and Joan L. Mitchell
3. *Implementing JPEG on the TMS320C6x*, application note
4. *TMS320C6211 Product Specification*, rev 1.31
5. *TMS320C6000 Chip Support Library API Reference Guide*
6. *TMS320C62x IMGLIB User's Guide*
7. *TMS320C6000 Optimizing C Compiler* (SPRU 187E)
8. *TMS320C6000 CPU and Instruction Set Reference Guide* (SPRU 190)
9. *TMS320C6000 Peripherals Reference Guide* (SPRU 190)
10. *The eXpressDSP Algorithm Standard* (SPRA 581)
11. *eXpressDSP Algorithm Standard Rules and Guidelines* (SPRU 352)
12. *eXpressDSP Algorithm Standard API Reference* (SPRU 360)

Appendix A. Controlling L2 Cache and DMAs Using CSL APIs

This is a brief summary on CSL API functions used in the JPEG encoder and decoder implementation. More details are found in [5].

L2 Cache APIs

CACHE_Clean: cleans a specific cache region

CACHE_EnableCaching: Enables caching for a specified block of address space

CACHE_Flush: Flushes a region of cache

CACHE_Invalidate: Invalidates a region of cache

CACHE_SetL2Mode: Sets L2 cache mode

DMA APIs (DAT APIs are architecture independent)

DAT_Close: Closes the DAT module

DAT_Copy: Copies a linear block of data from Src to Dst using DMA or EDMA hardware

DAT_Copy2D: Performs a 2-dimensional data copy using DMA or EDMA hardware

DAT_Fill: Fills a linear block of memory with the specified fill value using DMA or EDMA hardware

DAT_Open: Opens the DAT module

DAT_Wait: Waits for a previous transfer to complete

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgment, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its semiconductor products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Customers are responsible for their applications using TI components.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.