A MPEG Decoder in SHIM

[CSEE E6901.030 Project Final Report – December 2008]

Keerti Joshi
kj2217@columbia.edu

Delvin Kelleybrew
djk2125@columbia.edu

ABSTRACT

The emergence of world-wide standards for video compression has created a demand for design tools and simulation resources to support algorithm research and new product development. Because of the need for subjective study in the design of video compression algorithms it is essential that flexible yet computationally efficient tools be developed.

For this project, we plan to implement a MPEG standard using the SHIM programming language. The SHIM is a software/hardware integration language whose aim is to provide communication between hardware and software while providing deterministic concurrency.

The focus of this project will be to emphasize the efficiency of the SHIM language in embedded applications as compared to other existing implementations

INTRODUCTION

MPEG, which stands for Moving Picture Experts Group, is the name of a family of standards used for coding audio-visual information. It is a generic means of compactly representing digital video and audio signals for consumer distribution.

The major advantage of MPEG compared to other video and audio coding formats is that MPEG files are much smaller for the same quality. This is because MPEG uses very sophisticated compression techniques. Conventional compression algorithms can be divided into a sequence of stages: at best they can run in a parallel pipeline.

Implementing in SHIM could provide a good programming model for the MPEG, allowing it to be described concurrently and utilize SHIM to produce efficient parallel code. It follows a C-like syntax that allows local variable declarations and function calls, much like using a combination of C and an HDL such as Verilog.

In this project we plan to implement an existing algorithm of the MPEG decoder in SHIM and compare its complexity with other implementations of the video standard.

RELATED WORK

Image and video codecs are prevalent in multimedia devices, ranging from embedded systems, to desktop computers, to high-end servers such as HDTV editing consoles. It is not uncommon however that developers create and customize separate coder and decoder implementations for each of the architectures they target. This practice is time consuming and error prone, leading to code that is neither malleable nor portable. [5] paper describes an implementation of the MPEG-2 decoder using the StreamIt programming language, an architecture-independent stream language that aims to improve programmer productivity. The paper shows that MPEG is a good match for the streaming programming model and illustrates the malleability of the implementation using a simple modification to the decoder to support alternate color compression formats.

Efficient image processing techniques are needed to make images suitable for use in embedded systems. Alternatively, [6] describes an implementation of a JPEG decoder in the SHIM programming language.

MILESTONES

We divided the architecture of what it would take to perform the MPEG decoding into five modules:

1. *MPEG parser*
    - *General Input Parsing*
    - *Variable Length Decoding*
    - *Parser Outputs*
2. *Decoder split*
    - *"par" mechanism*
    - *Macroblock Decode*
    - *Motion Vector Decode*
3. *Motion Compensation*
4. *Scalability*
5. *Color Space Conversion*

MILESTONES ACHEIVED

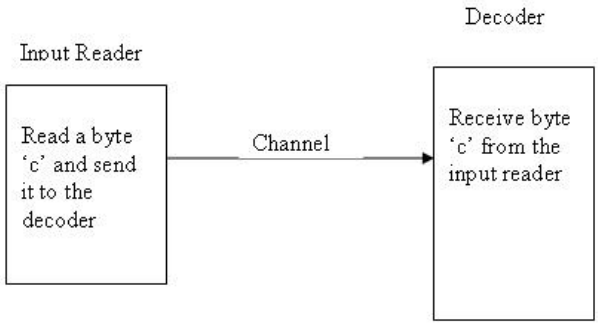
I.) MPEG Parser: The MPEG Parser performs the necessary decoding of the MPEG video file bitstream structure. A video sequence is made up of several main

layers (sequence, picture, slice, and macro-block layers). The purpose of the MPEG parser is to 1.) input bits/bytes from the MPEG bitstream, 2.) use unique identifiers called *startcodes* to find and differentiate between the main layers and there corresponding sub-layers, 3.) decode pertinent information from each layer and sub-layer, and 4.) output the fundamental elements needed to continue the MPEG decoding processes (i.e., macro-blocks containing information for luminance, chrominance, and motion vectors).

I.A) MPEG Parser – General Input Parsing: Before any processing of data could be done, a reliable method of retrieving this data from the MPEG file was devised. As previously mentioned, there were essentially two types of bitstream input: input describing the "data type" of the input to come (i.e., startcodes) and input used in the actual decoding process.

SHIM Implementation - The 32-bit code startcodes that are embedded in bitstream are unique and are used for several purposes including identifying some of the structures in the coding syntax. They follow a specific pattern that allows for easy detection of there position in the bitstream; every startcode is preceded by the 3-byte buffer of *0x00 0x00 0x01*. Knowing this, we were able to create a SHIM function called *get_startcode()* that searches that bitstream for the next startcode and returns its value:

```
void get_startcode(int32 &startcode)
{
    int32 c;
    while (1){
        c = getchar();
        if (c == 0x00){
            c = getchar();
            if (c == 0x00){
                c = getchar();
                while (c == 0x00){
                    c = getchar();
                }
                if (c == 0x01){
                    c = getchar();
                    startcode = c;
                    $$printf(" %x ", VAL(0)); $$
                    return;
                }
                else if(c == -1) return;
                else ;
            }
            else if(c == -1) return;
            else ;
        }
        else if(c == -1) return;
        else ;
    }
}
```



Because of SHIM's advantages with parallelism, an alternate *get_startcode()* function was also implemented that utilized the SHIM constructs *chan, send,* and *recv*. As seen in the figure above, these constructs allow for parallel computation in which *send* is able to transport streams of data on a channel (*chan) to* ***another*** module that contains a receiver (*recv).* There can only be one "sender" but theoretically, any number of "receivers". Using these constructs, we were also able to code an alternate *get_startcode()* function as such:

```
void read_file(chan int32 &c)

{

        int32 i = 0;
         while (1){
                $$ VAL (0) = (int) getchar (); $$
                 if (c == -1)
                 return;
                send c;
        }

}

void get_startcode(chan int32 c, int32
&buffer,int32 &start_code_cnt)

{

while(1){
      recv c;
      if (c==0x00)

{
  recv c;
```

When it came to retrieving decoder processing input, unlike with startcodes, there was no set scheme to follow since this data was not necessarily processed in "byte" patterns. At any given time during processing, the next 8 bits may be used for one information data point, followed by, say, the next 13 bits that may be used for another data point. Given this inconsistency in data processing, we were able to create a SHIM function call *get_bits()* which, given an integer, returns this many bits from the bitstream. Since bits can only be retrieved in "byte" groups, this function would retrieve 8 bits at a time and utilized a sliding window to sort of "mask" out the number of bits needed:

```
int32 get_bits(int number, int &bit_count)
{
  int i, newbit;
  int result = 0;
  int32 aux, wwindow, window;

  if (!number)
    return 0;

  for (i = 0; i < number; i++) {
    if (bit_count == 0) {
      wwindow = getchar();
      bit_count = 8;
    }
    else wwindow = window;
    newbit = (wwindow>>7) & 1;
    window = wwindow << 1;
    bit_count--;
    result = (result << 1) | newbit;
  }
  return result;
}
```

I.B) MPEG Parser – Variable Length Decoding: A variable length code is a way to map source symbols to a variable number of bits, allowing data compression. Unique Variable Length Code tables are utilized in various stages of the decoding process, including but not limited to macroblock addressing, macroblock type decoding, macroblock pattern detection, motion vectors, and DCT Co-efficients.

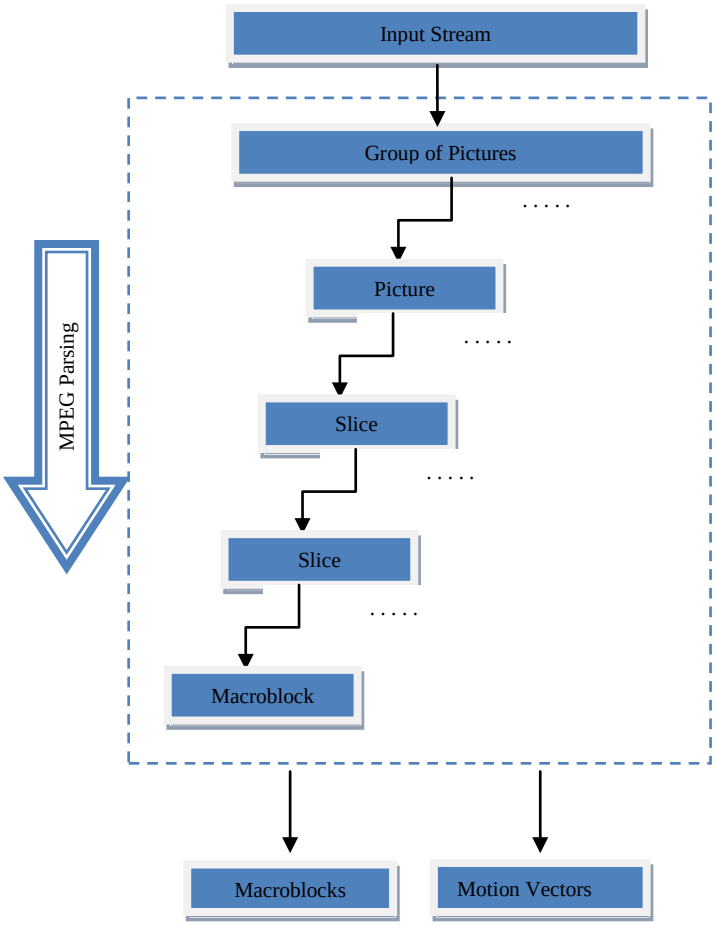
SHIM Implementation - However, since SHIM does not support pointers, the plan to implement this module was to generate a tree equal to the encoding tree and represent it using an array. Read an input bit and move to the left subtree if the bit is 0, the right subtree otherwise. . If a leaf is encountered, the value is returned. This "subtree" implementation can be seen as follows in a snippet of code from the macroblock addressing VLC function called *MBAVLC()*, which returns a macroblock address increment value given a VLC code:

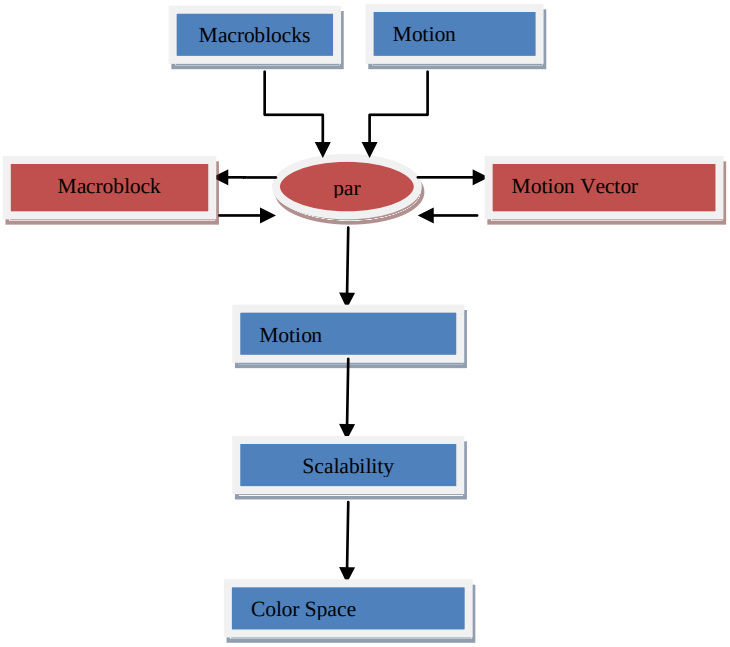
```
/*****************************************************/
// Function: MBAVLC
// Performs VLC coding for macroblock address (Table B-1 in standard)
/*****************************************************/
void MBAVLC(int32 &bits, int32 &bit_count, int32 &mbaIncVal)
{

  bits = get_bits(1,bit_count);
  if(bits ==1)
  {
    // VLC code 1
    mbaIncVal = 1;

  }
  else
  {
    // we have 0

    bits = get_bits(1,bit_count);
    if (bits == 1)
    {
      // we have 01

      bits = get_bits(1,bit_count);
      switch(bits){

      // VLC code 011
      case 0x01 : mbaIncVal = 2; break;

      // VLC code 010
      case 0x00 : mbaIncVal = 3; break;

      default: $$printf("ERROR - Invalid MBA VLC Code - Code Length 3\n");$$
      }
    }
    else
    {
      // we have 00

      bits = get_bits(1, bit_count);
      if (bits ==1)
      {
        // we have 001

        bits =get_bits(1,bit_count);
        switch(bits){

        // VLC code 0011
        case 0x01 : mbaIncVal = 4; break;

        // VLC code 0010
        case 0x00 : mbaIncVal = 5; break;

        default: $$printf("ERROR - Invalid MBA VLC Code - Code Length 4\n");$$
        }
      }
      else
      {
```

* * * *

The MPEG formal standard (ISO/IEC 13818-2) called for use of more than fifteen VLC tables in the scope of the entire decoding process, with some tables containing more than 200 elements. However, given our project time-constraints, we were able to implement approximately five of these tables.

I.C) MPEG Parser – Parser Outputs: Following the approach of the StreamIt implementation of the MPEG standard [5], the output of the parsed bitstream leads to two essential elements needed to further decode the video file: frequency encoded macroblocks and differentially coded motion vectors (see Figure below).

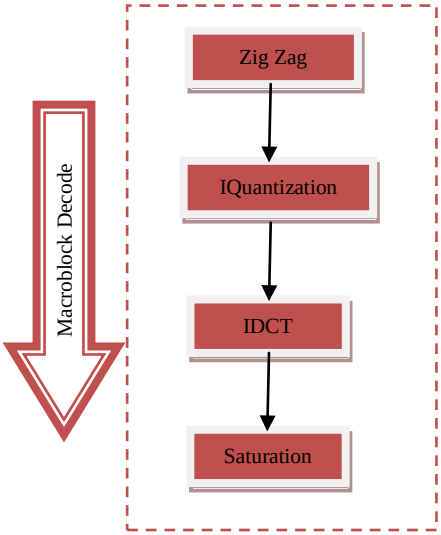




II.) Decoding Split: Once the essential outputs of the MPEG parser are calculated (macroblocks and motion vectors), the next level of processing begins to take advantage of the SHIM the mechanism for concurrent procedure calls known as the *par* statement. When utilized incorporation with the *next* keyword, this rendezvous-style inter-process communication follows the same approach as the *splitjoin* construct in the StreamIt language [5], allowing data to be distributed on parallel streams, and only continuing processing once both streams have returned.

IIA.) Decoding Split – *par* mechanism: The first use of the *par* mechanism could be implemented with the MPEG parser outputs as seen in the Figure below. Both the macroblocks and motion vectors have to undergo further processing that can, and should, be done in parallel.

IIB.) Decoding Split – Macroblock Decode: The frequency encoded macroblocks undergoe inverse transformations once passed on from the parser to the *par* mechanism.



Zig Zag: Zig zag scanning is a specific sequential ordering of the DCT coefficients from (approximately) the lowest spatial frequency to the highest.
Zig-zag descrambling is necessary to reorder the input stream generated by the run-length encoding of quantized DCT coefficients. Typically, the zig-zag scan operates on an 8x8 matrix. In MPEG, there are two possible scan orders. We represent the input matrix as a uni-dimensional stream of elements. The elements are then copied to the output stream in the specified order.

Inverse Discrete Cosine Transform (IDCT): The input stream is Huffman and run-length decoded, resulting in quantized DCT matrices. The DCT coefficients are scaled in magnitude and an inverse DCT (IDCT) is performed.

At this stage of the decoding process is where we began to see the core benefit of the SHIM language. Identifying the IDCT as a computation that can be done in parallel was the first step. However, time constraints did not allow us to write the code that would actually implement these parallel units. A description of this future work can be found the **Future Work** section of this report.

IDCT Parallelization: The number of parallel units can be hard-coded to a value k. If there are n blocks and they can be processed in parallel, then each thread becomes the owner of approximately n/k blocks. The computational speed hence expected is k times the speed of a single threaded system. Also, note that the n/k blocks of each unit are processed sequentially. At any given time there can be a maximum of k units running in parallel.

Saturation: Saturation limits a value that exceeds a defined range by setting its value to the maximum or minimum of the range as appropriate. The coefficients resulting from the Inverse Quantization Arithmetic are saturated to lie in the range
[-2048:+2047].

IIC.) Decoding Split – Motion Vector Decode: During the *par* split, the differentially coded motion vectors are is decoded to produce absolute motion vectors. The compression in MPEG is achieved largely via motion estimation, which detects and eliminates similarities between macroblocks across pictures.

For each macroblock, the motion estimator calculates a motion vector that represents the horizontal and vertical displacement of that macroblock from a similar matching macroblock-sized area in a reference picture. The matching macroblock is removed (subtracted) from the current picture on a pixel by pixel basis, and a motion vector is associated with the macroblock describing its displacement relative to the reference picture.

Future Work

I.) Continued Parser Output Processing: Completing the *par* mechanism usage for the "MacroblockDecode" and "MotionVectorDecode" splits; rejoining these units when complete for further processing.

II.) Motion Compensation: The motion compensation filter uses the motion vectors to find a corresponding macroblock in a previously decoded reference picture .The reference macroblock is added to the current macroblock to recover the original picture data. If the current macroblock is part of an I or P picture, then the decoder stores it for use as a future reference picture.

In the compensation stage, we plan to parallelize the processing of color channels The first handles the luminance color channel(Y), and the other two handle the chrominance channels (Cb and Cr)

III.) Scalability: This module pertains to the ability of the decoder to decode an ordered set of bitstreams to produce a reconstructed sequence. Useful video is output when subsets are decoded. The minimum subset that can be decoded is the first bitstream in the set, which is called the *base layer***.** Each of the other bitstreams in the set are called an *enhancement layer*. When addressing a specific enhancement layer, "lower layer" refers to the bitstream which precedes the enhancement layer.

- Different forms of scalability tools offered include *data partitioning, SNR scalability, spatial scalability, temporal scalability,* and *hybrid scalability*.

III.) Color Space Conversion: MPEG is targeted for a set of specific applications; therefore there is only one color space that corresponds to it – *4:2:0 YCbCr*. This module, which is the last stage of the MPEG decoding process, is responsible for a linear transform from the YCbCr color space to the R0G0B0) color space. That is, gamma-corrected red, green, blue are computed from a luminance-related quantity called luma (Y), and two color difference components called chroma (Cb and Cr).

IV.) Future Work Overview: Below is a depiction of a top-level view of the MPEG decoding sequence, illustrating what the final SHIM model of the decoder looks like once all milestones are achieved and modules implemented.

Conclusion

With this project, we encountered a few SHIM language constraints that dictated many of our approaches in design and coding. These constraints included no use of global variables, pointers, or dynamic allocation. There was even a limit on the integrity of data structures since SHIM currently does not allow an individual element of an array or a struct to be passed by reference (e.g., the entire array or struct has to be passed by reference).

However, despite these limitations, we did come across beneficial aspects to using SHIM versus other streaming languages. When it came to employing any of the parallel constructs of SHIM, it was easier transporting data across different modules, as well as controlling this data and its manipulation during processing in different layers. Particularly, *chan, send,* and *recv* constructs allowed for reassurance that a new stream of bits would not be

transported to until the receiving modules “unlocked” their hold on the previous stream.

Convenient file handling was another benefit of other languages so no hard coding file names or pointers was necessary; instead, all file streaming can be handled at runtime in the command prompt.

At the milestone reached MPEG decoder, we have approximately 5200 lines of SHIM code, whereas the current corresponding SHIM JPEG decoder has approximately 1000 lines of SHIM code. This comparison shows the SHIM compiler and language seems to be robust and reliable. Overall, the project helped to get a good understanding of the capabilities of SHIM and its usage.

REFERENCES

[1] SHIM: A Deterministic Concurrent Language for Embedded Systems Presented at Princeton University, New Jersey, May 10th, 2007

[2] “SHIM: A Scheduling-Independent Concurrent Language for Embedded Systems.”Presented at the University of California, Berkeley, Berkeley, CA, November 8th, 2006.

[3] Didier Le Gall “MPEG: A Video Compression Standard for Multimedia Applications” Trans. ACM, April 1991.

[4] ISO/IEC 13818-3:1998. Information technology – Generic coding of moving pictures and associated audio information. Available at http://www.iso.org.

[5] Amarasinghe, Saman, Matthew Drake, Hank Ho_Mann, and Rodric Rabbah. “MPEG-2 Decoding in a Stream Programming Language.” Computer Science and Artificial Intelligence Laboratory. Massachusetts Institute of Technology

[6] Stephen A. Edwards and Nalini Vasudevan. “A JPEG Decoder in SHIM.” Department of Computer Science. Columbia University