MPEG2Event: A Framework for Developing Analysis Tools for Compressed Video

Ketan Mayer-Patel
University of North Carolina, Chapel Hill

Jim Gemmell
Microsoft Research

October 2004

Technical Report
MSR-TR-2004-111

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

MPEG2Event: A Framework for Developing Analysis Tools for Compressed Video

Ketan Mayer-Patel
University of North Carolina, Chapel Hill

Jim Gemmell
Microsoft Research

October 2004

Abstract

This paper describes two contributions to the analysis of MPEG video compression: the MPEG2Event library and the MPEGstats web site. MPEG2Event is a C# library intended to facilitate rapid prototyping of MPEG-2 analysis tools. Unlike other MPEG-2 decoding libraries that are designed for performance, MPEG2Event sacrifices parsing speed in order maximize flexibility and expose the coding elements contained within an MPEG-2 stream at a number of different granularities. MPEGstats is a web site/service that makes coding statistics about DVD titles available to the worldwide research community. The MPEGstats web site has a database backend, populated with the statistics of DVDs, as extracted by the MPEG2Event library. The user interface of the web site allows common queries to be performed, e.g. by title and chapter, and/or by certain frames types. A .NET web service is also implemented to allow customized access to the data.

1. Introduction

This paper describes two contributions to the analysis of MPEG video compression: the MPEG2Event library and the MPEGstats web site. MPEG2Event is a C# library intended to facilitate rapid prototyping of MPEG-2 analysis tools. MPEGstats is a web site/service that makes coding statistics about DVD titles available to the worldwide research community. The motivation of this work is to make real world MPEG statistics more readily available to video researchers. In our own research, we have been struck by the scarcity of tools available for deconstructing and analyzing MPEG-2 video. While many open source implementations of the codec exist, none provide the ability to catalog and analyze the specific coding elements within the stream. We realized that although many researchers purport to be exploring new ways to transmit, store, and use compressed video (and often MPEG specifically), they rarely model the video sources more accurately than by characterizing its bitrate.

Unlike other MPEG-2 decoding libraries that are designed for performance, MPEG2Event sacrifices parsing speed in order maximize flexibility and expose the coding elements contained within an MPEG-2 stream at a number of different granularities. Our main motivation in designing MPEG2Event was to provide researchers with a way to answer questions about the structure of an MPEG-2 compressed video stream. Thus, it was less important to provide facilities for reconstructing actual pixel values. This is the biggest difference between our library and currently available codecs and explains why existing codecs fall short of what we needed.

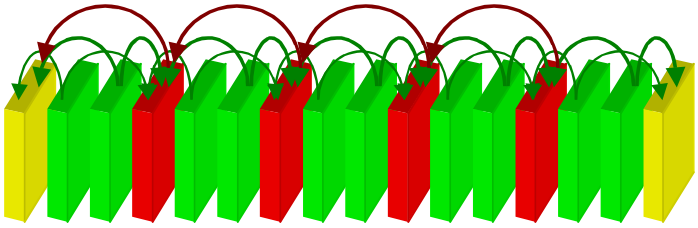
The MPEGstats web site has a database backend, populated with the statistics of DVDs, as extracted by the MPEG2Event library. The user interface of the web site allows common queries to be performed, e.g. by title and chapter, and/or by certain frames types. A .NET web service is also implemented to allow customized access to the data.

In the remainder of this paper, we begin with a basic review of the MPEG encoding scheme in order to give the reader a sense for the syntactic and semantic complexity within a compressed stream. Section 3 then describes the MPEG2Event library, with a sample analysis program, and a discussion of performance

issues. The MPEGstats web site is covered in Section 4. Section 5 recounts our experience testing a hypothesis about video coding using MPEG2Event, followed by our conclusion and plans for future work.

2. Review of MPEG encoding

An MPEG-2 video sequence is comprised of a sequence of frames [1]. Each frame is encoded in one of three ways resulting in three different types of frames labeled I, P, and B. I-frames are encoded in a manner that is completely self-contained. Thus, given the bits corresponding to an I-frame, the frame can be completely recovered. P-frames employ motion compensation as a technique using the previous I- or P-frame as a prediction basis. Thus, P-frames depend on information contained in the previous I- or P-frame. As P-frames depend on each other, a dependency chain is created between an I-frame and any subsequent P-frames until the next I-frame is encountered. B-frames employ motion compensation in two directions, using both the previous I- or P-frame as well as the subsequent I- or P-frame. B-frames, however, are not used by any other frame as a prediction basis. The pattern of frame types employed is completely ad hoc and there are very few restrictions. A typical pattern, however, that is employed by many encoders is illustrated in Figure 1. In this figure, each frame in the sequence is labeled by its type and arrows indicate dependency relationships. An obvious problem with B-frames is that they depend on a frame from the future. Since decoding B-frames can not be done until both its past and future reference frames (i.e., previous I- or P-frame and next I- or P-frame) are decoded, frames are not sent in display order, but rather in decode order. This is also illustrated in Figure 1.



Type:	**I**	**B**	**B**	**P**	**B**	**B**	**P**	**B**	**B**	**P**	**B**	**B**	**P**	**B**	**B**	**I**
Display Order:	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Encoded Order:	**0**	**2**	**3**	**1**	**5**	**6**	**4**	**8**	**9**	**7**	**11**	**12**	**10**	**14**	**15**	**13**

Figure 1: MPEG Frame Types And Dependencies

Each frame, regardless of type, contains three planes of pixel information. One plane contains luminance information (i.e., the grayscale value of the pixel) and two planes provide chrominance information (i.e., color). These planes are often labeled Y, U, and V with Y being the luminance plane and U and V referring to the two chrominance planes. Although we will adopt this notation for convenience, the reader is cautioned that these labels have very specific technical meanings within the domain of analog and digital video standards. These planes are not equal sized. The U and V planes are subsampled by a factor of two in both directions. Each plane is organized into tiles of 8x8 blocks. Given the subsampling of the U and V planes, each block of the U and V planes corresponds to 4 blocks of the Y plane. Each 2x2 group of blocks from the Y plane and their corresponding block from the U and V planes form a macroblock. The frame is encoded as a sequence of macroblocks organized in row-major order. This is illustrated in Figure 2.

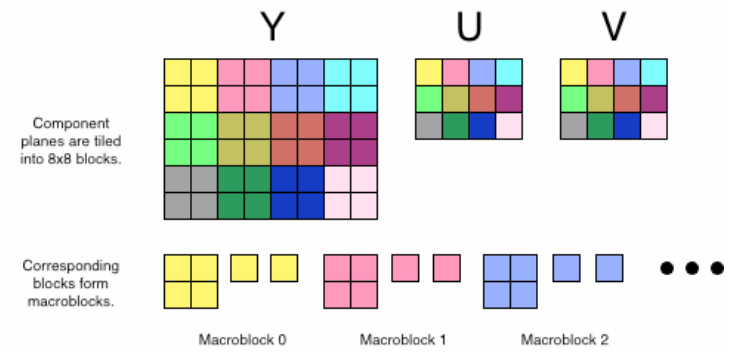


Figure 2: Organization of Pixel Data into Macroblocks.

Within a macroblock, each block of pixels (i.e., the four Y blocks, the U block, and the V block) is encoded as a quantized set of DCT coefficients. If motion compensation is employed, the macroblock is also associated with one or more motion vectors that are used in conjunction with the previous and/or next I- or P-frame to from a prediction basis. In this case, the difference between the prediction basis and the pixel values is encoded by the DCT instead of the actual pixel values. In either case, the coefficients are quantized using a scale factor that is either explicitly specified within this macroblock or implicitly defined as the same scale factor in effect for the previous macroblock. In this way, each encoded macroblock is a collection of a number of different encoded elements. These elements include mode information indicating the presence or absence of other elements, motion vectors used to form a prediction basis from previous or future frames, a quantization scale factor, DCT coefficients, etc. Table 1 describes many, but not all, of the coding elements that may be found within a macroblock.

Table 1: Component Coding Elements of a Macroblock

Coding Element	Description
Address Delta	The difference between the address (i.e., position) of this macroblock and the previously encoded macroblock. Values greater than 1 indicate the presence of 1 or more “skipped” macroblocks.
Mode	A variable length code that indicates the presence or absence of the quantizer scale code, motion vectors, and block pattern.
Quantiser Scale Code	If present, sets the quantization factor.
Motion Vectors	If present, indicates the offset between the current macroblock position and the position of the prediction basis within a reference frame. Up to 4 motion vectors may be present depending on the type and mode of motion compensation in use.
Coded Block Pattern	If present, indicates which of the 6 blocks of the macroblock are actually encoded. Some blocks may not be encoded if the difference between the pixels values and the prediction basis is sufficiently small.
Blocks	The encoded DCT coefficients of that make up this macroblock. This element is further comprised of individual elements that encode each coefficient.

Macroblocks are organized into slices. Typically each slice is one row of macroblocks, but this is not required. A slice header that resets certain key decoder state variables introduces each slice. A picture header and a picture coding extension header introduce the slices that comprise one frame. These headers indicate the type of the frame (i.e., I-, P-, or B-), its temporal relationship to other frames, and other pieces of information that affect how coding elements within the frame such as motion vectors are decoded and interpreted. The separation of this information into a picture header and a picture coding extension header is required to provide backwards compatibility with MPEG-1. Pictures are grouped together to form a GroupOfPictures (GOP). A GOP header introduces each GOP. Each GOP is typically comprised of an I-frame and all subsequent frames until but not including the next I-frame, although this is not mandated by the standard. The GOP header primarily provides a timecode used for synchronization. The entire stream is introduced by a sequence header and a sequence extension header that provides stream level information such as the frame rate, frame dimensions (i.e., width and height), and other information that remains static for the entire lifetime of the stream. The sequence header must exist at the beginning of the stream, but may be repeated any number of times during the stream. If repeated, however, the information must be identical to previous instances of these headers and they can only precede a GOP header. The overall organization of the MPEG-2 bitstream is illustrated in Figure 3.

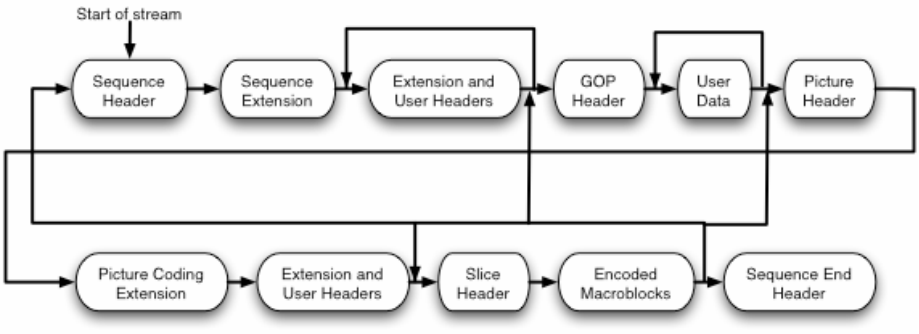


Figure 3: MPEG2 Bitstream Organization

This brief overview of the MPEG-2 bitstream is far from complete. Myriad details about specific coding elements and techniques have been omitted for clarity. The overview serves to provide the reader with a sense of the overall organization of the bitstream as well as to illustrate that the MPEG-2 bitstream is quite complex with hundreds of different coding elements organized into many different syntactic layers. This complexity motivates much of the design of our analysis library.

3. MPEG2Event

In this section, we motivate and describe the overall design of MPEG2Event. We then provide a more detailed outline for how the MPEG2Event library can be used, highlighting the class hierarchy of coding elements and the event mechanisms that are the central features of the framework. An example analysis program is described as a tutorial in order to make these concepts more concrete. Finally, performance issues are explored and the results of some basic experiments quantifying performance are given.

3.1. Design goals for MPEG2Event

Existing codecs are designed primarily for displaying decoded video in real time. As a result, many operations of the decoding process are folded into each other in an attempt to optimize the decoding process. For example, dequantization of DCT coefficients and the inverse DCT process can be accomplished within the same inner loop. If the processor provides a multimedia-specific instruction set such as MMX, the dequantization step can be folded into the inverse DCT algorithm almost completely. Also, many of the coding elements within an MPEG-2 video stream only influence the decoding process of nearby coding elements and therefore do not affect global decoder state. Within existing codecs, these elements are decoded and their values stored and used only within the local scope of the decoding subroutine where they are encountered and required. Modifying an existing decoder to capture information about these coding elements would require hunting down every subroutine within which these coding elements may be encountered and instrumenting these subroutines to report their values. To do so would be tedious and error-prone, especially with a codec that has been heavily optimized for performance.

The overall design goals for our library include:

- Coding elements should be exposed at a number of different granularities allowing tool builders to access information at the level most congruent with the needs of the user.
- Every bit of the stream should be parsed and available as part of some coding element.
- A tool should be able to effectively ignore any and all coding elements that are outside of the user's interests.
- A distinction should be made between coding elements that represent atomic units of data and coding elements that are comprised of a collection of sub-elements.
- The information encoded in the stream should be available in its most basic form as encoded.

Given these goals, the design of MPEG2Event is event-based, hence the name of the library. As the video stream is parsed, each atomic coding element encountered is published as an event using the C# multicast event mechanism. Collecting together the component elements that comprise these structures forms higher-

level syntactical structures such as a block of DCT coefficients, the elements of a macroblock, the macroblocks of a slice, a particular header, and so on. The higher-level structures are similarly published as events.

Information about each of these events is encapsulated into an instance of a corresponding C# class which represents the event. These event types are organized into a class hierarchy allowing one to subscribe to events in a number of different ways including:

- Subscribing to a very specific atomic coding element (e.g., the horizontal size field in the picture header, the vertical component of a motion vector, etc.).
- Subscribing to a compound coding element comprised of smaller coding elements (e.g., the picture header as a whole, a motion vector, a macroblock, etc.).
- Subscribing to a general class of events (e.g., all headers, all atomic coding elements, etc.).

By employing the C# multicast event mechanism, any number of callbacks can be registered to handle any specific event type. This facility allows a particular coding element event to be processed by two or more different callbacks for independent purposes. Furthermore, callbacks can be registered or unregistered dynamically. This allows tools to be built that conditionally process different coding elements based on the contents of the stream.

In most applications, MPEG-2 video data is multiplexed with audio and auxiliary information in either an MPEG-2 program or MPEG-2 transport stream. Currently, MPEG2Event only works with MPEG-2 elementary video streams. Other tools must be used to demultiplex the video stream out from within a transport or program stream. The MPEG2Event architecture, however, is very general and could be easily extended to work with transport, program, or audio streams.

3.2. Basic Usage

For most uses of the library, the following basic approach can be followed:

- Wrap the source of bits into a BitStream object.

  The BitStream class provides a bit parser interface to any C# Stream object. Simply pass the source of the MPEG-2 stream data (i.e., file stream, network stream, etc.) as a parameter to the BitStream constructor.

- Register delegate methods as callbacks for the coding elements that you would like to process.

  The callback method signature must match the delegate definition found within the coding element event class. Register the callback by adding the delegate to the statically defined "handlers" event for each event class.

- Create a VideoParser object to parse the video.

  The VideoParser does the work of actually parsing the video stream. It does so by keeping track of decoding state and knowing what syntactic structure to expect next. It uses the BitStream object created above to retrieve the actual bits from the stream.

- Call the parse_picture() method of the VideoParser object in an infinite loop (or for however many pictures you want to parse).

  This method will throw an exception if an error in the bitstream is found or if the end of the stream is reached.

- As coding elements are parsed, events are constructed and published to registered callback methods as appropriate.

From this general skeleton, we can see how MPEG2Event provides the tool builder with a flexible analysis framework that allows the developer to focus on his goals instead of worrying about the details of decoding. Only the events of interest need to be dealt with and all information about a particular coding element is encapsulated into the event object passed to the handler.

3.3. Coding Element Event class hierarchy

The coding element event classes that encapsulate information about each coding element encountered is organized into a relatively flat hierarchy. This hierarchy allows one to subscribe to more general abstract classes of coding events that may encompass several specific subtypes. At the root of the coding element class hierarchy is the "CodingElement" class. This is an abstract class with no direct instantiations. This class encapsulates information common to every coding element including:

- BitAddress

  A read only property returning the bit count of the first bit of this coding element. In other words, the number of bits read from the BitStream object before this coding element was encountered. NOTE: this is the count of bits not bytes.

- Length

  A read-only property returning the length of this coding element in bits as encoded in the bitstream.

- isContiguous()

  A method that returns a boolean value indicating whether all of the bits that encode this coding element are contiguous in the bitstream. The was intended to support coding elements which are compound elements which encapsulate other coding elements that are not necessarily back-to-back within the bitstream. In practice, all coding elements, including compound ones like the picture header, are contiguous.

3.4. Atomic vs. Compound Coding Elements

At this point, it may be useful to discuss the difference between an "atomic" coding element and a "compound" coding element. An atomic coding element is a sequence of bits that encode a specific value with a specific purpose. For example, a 32-bit start code is an atomic coding element, the 10 bits that make up the temporal reference number in a picture header is an atomic coding element, and so on. A compound coding element is a collection of one or more atomic coding elements that form a higher level of syntax for the video stream. For example, all of the atomic coding elements that go into a picture header are collected together in a compound coding element called PictureHeader. Atomic coding elements are published as events as they are encountered. Compound coding elements are published after the last of its components is published.

Compound coding element events generally provide an interface to retrieve its component parts if you need them. This allows you to subscribe to the level of detail that makes the most sense. If, for example, you are interested in a number of different pieces of information that all reside in the picture header (e.g., the temporal reference, the picture coding type I, P, or B, etc.), then it would make most sense to subscribe to the entire picture header event and then get to the specific pieces of information you need by going through the published picture header event object. If you really just need the temporal reference, then it would make more sense to just subscribe to the temporal reference event.

Atomic coding element events are all subclasses of the abstract class AtomicCodingElement, which in turn is a direct subclass of CodingElement. Most compound element events are direct subclasses of CodingElement. The notable exceptions are the SequenceHeader, SequenceExtension, PictureHeader, PictureCodingExtension, GroupOfPicturesHeader, SliceHeader, and QuantMatrixExtension, which are all grouped together as subclasses of Header that in turn is a direct subclass of CodingElement. The Header class does not provide much functionality other than give you a convenient way to subscribe to all the header type events without having to specify a handler for each specific header type. Note that the start code element that announces the presence of a particular header is not considered part of the header (i.e., the StartCode atomic coding element is not contained by the header that the StartCode identifies).

It is important to note that the hierarchy of these coding element event classes does not reflect the organization of the MPEG-2 bitstream. In other words, the Slice coding element encapsulates the Macroblock coding elements generated by the macroblocks in that slice. These Macroblock coding element event objects in turn encapsulate MacroblockAddressDelta, MacroblockMode, and Block coding elements.

However, all of these coding elements event types mentioned are simply direct subclasses of CodingElement since they are all compound coding elements. For more information on the organization of an MPEG-2 bitstream (i.e., which compound elements contain which other compound and atomic coding elements, what order things arrive in, etc.) the reader is referred to the MPEG-2 standard.

Figure 4 shows an abbreviated map of the coding element class hierarchy. The numerous atomic coding elements are not all individually listed, but it should provide a general idea for how things are structured and lists some of the more important atomic coding elements that are likely to be useful.

- CodingElement
  - AtomicCodingElement
    - StartCode
    - OpaqueBits
    - HuffmanEncodedCoeff
    - TemporalReference
    - PictureStructure
    - PictureCodingType
    - Other subclasses of AtomicCodingElement too numerous to list individually.
  - Header
    - SequenceHeader
    - SequenceExtension
    - PictureHeader
    - PictureCodingExtension
    - GroupOfPicturesHeader
    - SliceHeader
    - QuantMatrixExtension
  - Macroblock
  - MacroblockAddressDelta
  - MacroblockMode
  - MotionVector
  - Block
  - DiffEncodedCoeff
  - MatrixCodingElement
    - IntraQuantiserMatrix
    - NonIntraQuantiserMatrix
    - ChromaIntraQuantiserMatrix
    - ChromaNonIntraQuantiserMatrix
  - UserData
  - Slice

Figure 4: Abbreviated Hierarchy of Coding Element Event Classes

3.5. Event Mechanism

Each class in the coding element event class hierarchy implements an event-based mechanism for informing your code via callbacks when a particular coding element has been parsed. To explain how this mechanism works, we take a closer look at how it is implemented in the root CodingElement class.

The CodingElement class defines a delegate with the following signature:

```
public delegate void Handler(BitStream bs, CodingElement e)
```

This is the signature that the callback must adhere to. The first argument, *bs*, will be set to the BitStream object that the coding element was parsed from. This is useful for processing more than one stream at a time as it allows the handler to disambiguate coding elements from different streams. The second argument, *e*, is set to the event object that encapsulates the coding element parsed.

Given a callback with the appropriate signature, a delegate for the callback is constructed in the normal C# way. The delegate serves as a type-safe wrapper for the function pointer to the callback. Once the delegate is defined, the callback is registered with the statically defined multicast event in the coding element class named “handlers." When a coding element is parsed, it publishes itself to all of the registered callbacks by invoking the virtual protected method "publish". This method first publishes the event to any parent classes by invoking the publish method in its base class. In this way, any and every callback registered with a particular coding element class or any ancestor of that class will be called with the more general handlers being invoked before the more specific handlers.

Every subclass in the hierarchy under CodingElement implements this publication mechanism and by convention, each defines the delegate as "Handler" and maintains the registered callbacks in the statically defined multicast event "handlers." For example, the coding element event class PictureCodingType, which

is instantiated when the frame type (i.e., I, P, or B) is encountered in the picture header, defines the PictureCodingType.Handler delegate and maintains registered callbacks in the class variable PictureCodingType.handlers.

3.6. The Basic Video Parser

Each class in the coding element event hierarchy implements a static class method called "GetNext(BitStream bs)". This method assumes that the next bit in the bitstream passed as a parameter is the first bit of the coding element. The method then parses the coding element, constructs a new instance of the coding element event, and publishes the event to all registered handlers. While each coding element event class knows how to parse an instance of itself, determining the order of the coding elements to be parsed and managing the video parsing process is not provided by these methods per se. A helper class called VideoParser is provided for this task.

A VideoParser object provides a very simple public interface. The constructor requires a BitStream object to be passed as a parameter. This is the source of all bits for the stream being parsed. If more than one stream needs to be parsed, a separate VideoParser must be instantiated for each stream. A VideoParser object provides only two public methods which are:

- public void parsePicture();
- public void skipPicture();

These methods parse the next picture or skip the next picture, respectively. When either is called for the first time, the parser skips over all bits until it encounters a MPEG-2 sequence header. These skipped bits are published as OpaqueBits events. The sequence header, sequence extension header, any user data and other extension headers, group of pictures header, picture header, and picture coding extension header are then parsed and published. If the picture is to be parsed (as opposed to skipped), the picture data (i.e., slices of macroblocks) are parsed and published. If the picture is to be skipped, slice headers are parsed and published and all of the coding elements between the slice headers (i.e., the actual coded picture information) are published as OpaqueBits. Parsing ends at the end of the picture data. Subsequent calls to either method begin parsing where the last call left off.

The skipPicture method is useful to avoid parsing and publishing the coding elements of a picture that is known not to be needed. For example, if only I-frames are of interest, skipPicture can be used to avoid parsing P and B frames when encountered. After the sequence end code has been encountered, both methods will simply return without doing anything. This condition can be checked using the "EOF" property of the parser object. If the bitstream runs out of bits, the underlying BitStream object will throw an exception that can be caught to detect this situation.

3.7. Example: Collecting Rate-Distortion Information

In this example, rate distortion information is collected from the video stream. The goal is to map the relationship between the number of DCT coefficients encoded in a macroblock and the size of the encoded macroblock in bits. Macroblocks are accounted for separately by the type of motion compensation in effect. Intra macroblocks are macroblocks that do not use any sort of motion compensation. While all macroblocks of an I-frame are intra macroblocks by definition, these macroblocks may also appear in P or B frames. Forward macroblocks are those that employ forward motion compensation (i.e., all non-intra blocks of a P-frame and possibly some non-intra blocks from a B-frame). Backward macroblocks employ backward motion compensation and bidirectional macroblocks employ both backward and forward motion compensation. These macroblocks can only appear in B-frames.

The main body of code is listed in Figure 5. Lines 3-11 declare variables for storing the sum of macroblock lengths given a particular number of coefficients in the macroblock and the number of macroblocks that contributed to this sum. In the main routine, Line 17 instantiates a new BitStream object to read the video stream bits from a file and Line 19 creates a new VideoParser to parse those bits and generate coding element events. As we are interested in information that is available at the macroblock level, Line 21 creates a new delegate for handling Macroblock events and registers the delegate with the multicast event. Lines 24-32 form the main loop in which the video parser is simply instructed to parse pictures until either the sequence ends or the file runs out of bits. Finally, the results are printed. The body of the print_results()

subroutine is omitted. Lines 37-67 define the subroutine that handles Macroblock events which was used to define the delegate registered with the Macroblock event. In that subroutine, Lines 40-49 iterate over Block coding elements encapsulated by the Macroblock coding element in order to count the number of coefficients each contains. The length of the Macroblock as encoded in the bitstream is retrieved in Line 51. Lines 53-66 use properties of the MacroblockMode element to classify the block and update the counters appropriately.

```
1       public class RateDistortionExample {
2
3         static private ulong[] i_mb_size = new ulong[64*6];
4         static private ulong[] f_mb_size = new ulong[64*6];
5         static private ulong[] b_mb_size = new ulong[64*6];
6         static private ulong[] bi_mb_size = new ulong[64*6];
7
8         static private ulong[] i_mb_count = new ulong[64*6];
9         static private ulong[] f_mb_count = new ulong[64*6];
10        static private ulong[] b_mb_count = new ulong[64*6];
11        static private ulong[] bi_mb_count = new ulong[64*6];
12
13        public static void Main(String[] args) {
14
15          String filename = args[0];
16
17          BitStream bs = new BitStream(File.OpenRead(filename));
18
19          VideoParser vp = new VideoParser(bs);
20
21          Macroblock.handlers += new Macroblock.Handler(count_coefficients);
22
23          int picture_count = 0;
24          try {
25            while(!vp.EOF) {
26             vp.parsePicture();
27             System.Console.WriteLine(picture_count++);
28            }
29          }
30          catch (OutOfBitsException e) {
31            System.Console.WriteLine("Ran out of bits\n");
32          }
33
34          print_results();
35        }
36
37        public static void count_coefficients(BitStream bs, Macroblock mb) {
38
39          int coeff_count = 0;
40          foreach (Macroblock.BlockIndex idx in
41                  Enum.GetValues(typeof(Macroblock.BlockIndex))) {
42            Block b = mb.GetBlock(idx);
43            if (b != null) {
44             coeff_count += b.ACCount;
45             if (!b.DC.IsImplicit) {
46               coeff_count++;
47             }
48            }
49          }
50
51          int length = mb.Length;
52
53          MacroblockMode mode = mb.Mode;
54          if (mode.Intra) {
55            i_mb_size[coeff_count] += (ulong) length;
56            i_mb_count[coeff_count]++;
57          } else if (mode.MotionBwdSet && mode.MotionFwdSet) {
58            bi_mb_size[coeff_count] += (ulong) length;
59            bi_mb_count[coeff_count]++;
60          } else if (mode.MotionBwdSet) {
61            b_mb_size[coeff_count] += (ulong) length;
62            b_mb_count[coeff_count]++;
63          } else {
64            f_mb_size[coeff_count] += (ulong) length;
65            f_mb_count[coeff_count]++;
66          }
67        }
```

Figure 5: Example code using MPEG2Event to map the relationship between the number of coefficients in a macroblock and the length of the macroblock in bits as encoded.

This example shows how analytical tools can be rapidly developed with MPEG2Event because it allows the developer to concentrate on the coding elements of interest and access them at the appropriate granularity.

In this case, the information required existed in coding elements that are encapsulated at the macroblock level. Thus, we are able to write a single event handler for Macroblock events and through that handler retrieve information from component coding elements. In addition to the value of various coding elements (e.g., the actual coefficients in the macroblock), these elements provide information about their encoding as well (e.g., the length of the macroblock in bits as encoded). This is information that is usually not preserved in more traditional codec architectures.

Although the example here serves to illustrate the ease of use and flexibility of the library, the results of running the program on the twelfth chapter of a DVD are graphed in Fig. 6 for the interested reader. Of particular interest is the ability for number of coefficients to serve as proxy for encoding rate when the number of coefficients is fewer than 150. The mostly linear relationship between the number of encoded coefficients and the encoded size of the macroblock regardless of motion compensation type suggests that this can be effectively used within a rate control algorithm for dynamically adjusting quantization to achieve a particular rate target. However, the highly non-linear relationship when the number of coefficients is larger than 150 indicates the limits of this rate proxy. Even more interesting is the seemingly counter-intuitive relationship between number of coefficients and encoded size for intra macroblocks when the number of coefficients is in the range of 150 to 200. The results would seem to indicate that fewer bits are needed to encode a larger number of coefficients. We hypothesize that the reason for this anamoly is that because the coefficients are run length encoded and since pairs of runs and values are then Huffman encoded, this range represents a particular distribution of runs for which particularly small Huffman codes exist, thus allowing more coefficients to be encoded by fewer bits. A more thorough analysis would need to be completed to confirm this hypothesis, but it demonstrates quite well how the MPEG2Event framework can provide researchers with the tools for doing just this sort of analysis.

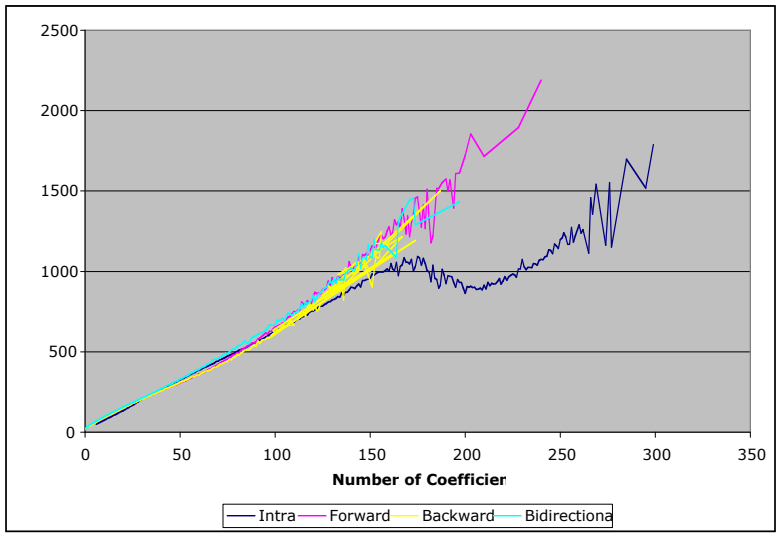


Figure 6: Results of example program run on one chapter of the DVD "Two Week's Notice".

3.8. Performance

Because of its unique event-based architecture and the manner in which it exposes coding elements at many levels of granularity (i.e., both as individual atomic elements as well as compound elements that group and encapsulate syntactically related items), the library can not be expected to compete against highly optimized codecs in terms of decoding performance. This is price that must be paid in order to reap the benefits of rapid prototyping and flexibility. In this section, we present the results of a simple experiment to measure the performance of the library in order to quantify this cost.

In this experiment, we wrote a program using the library that counted the number of atomic coding elements in a video stream. To do this, we registered a simple callback with the AtomicCodingElement class and simply incremented a counter whenever an event was delivered to the handler. Because the AtomicCodingElement class is the root of all atomic coding elements, every bit of the video stream is accounted for. Although we do not register any handlers for any compound coding element events, these

elements are still constructed during the parsing operation. This represents a possible future optimization. If the analysis of registered callbacks can guarantee that the element being constructed in not of interest to any other part of the code, it may be possible to avoid the overhead of object instantiation. Doing so, however, is not a trivial task and complicates the design of the library.

The results of the experiment are presented in Figures 7 and 8. The experiment was performed on one chapter of a single DVD. Other experiments have shown these results to be representative of decoding speed in general. All of these experiments were conducted on a Pentium 4 CPU running at 3.0 Ghz with 1GB of RAM. Figure 7 shows the performance of MPEG2Event as measured in the number of coding elements parsed and counted per second. Performance is fairly uniform at around 3600 elements per second. Figure 8 shows the same information presented as bits per second. This measure of performance is more variable since much of the work MPEG2Event does is on a per coding element basis but different coding elements have different bit lengths. At 15kbps, the library is about a factor of 200 times slower than real-time.

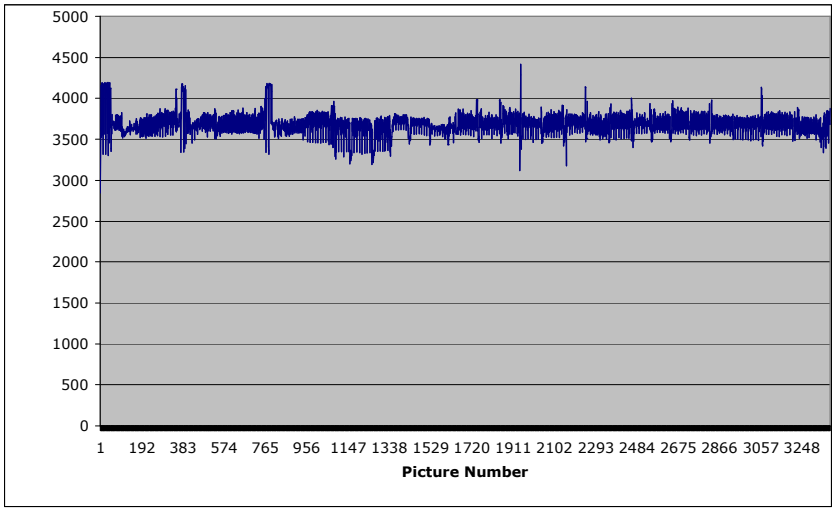


Figure 7: Decoding speed measured in elements per second for one chapter of a DVD.

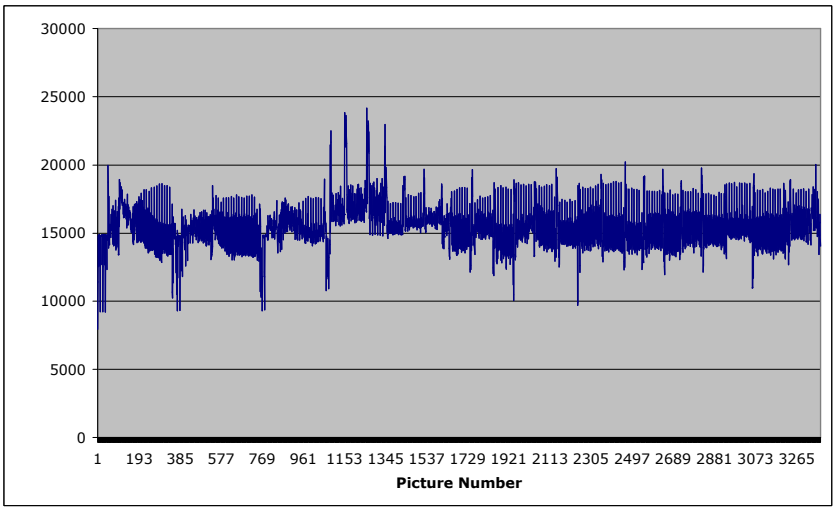


Figure 8: Decoding speed measured in bits per second for one chapter of a DVD.

4. MPEGstats

MPEGstats is a website/web-service that makes frame-level statistics fro DVD titles available to the worldwide research community. We plan to collect statistics from a broad spectrum of titles and genres into

a backend database and develop a web service front end through which other researchers can browse and retrieve these statistics. In this section we motivate why such a service will be useful and is needed by the research community, describe the schema associated with the statistics gathered, and briefly describe the envisioned web-accessible front end. The section concludes with a few thoughts about possible legal and copyright issues and our planned approach for dealing with them.

4.1. Motivation and Impact

Although compressed video is highly structured with complex coding dependencies among different constituent elements, few researchers incorporate this structure into their models of the data type when evaluating experimental systems and streaming protocols. Often video is modeled simply by bitrate and ad hoc rate-distortion curves with little to no empirical basis. In the case of MPEG, frame types (i.e., I, P, and B) and inter-frame coding dependencies are sometimes included to more realistically model the effect of packet loss, but even then the frame type pattern is often assumed to be static and does not necessarily reflect patterns in actual use. We believe that given better information about how commercial video data is actually being compressed, researchers will be able to build more realistic models of video data and thus more realistically evaluate experimental systems and protocols.

In order to provide this information, two challenges must be addressed. First, we need methods for extracting and analyzing information about how video data is being compressed. Second, we need an accessible, central repository for this information. The first of these challenges is directly addressed by the design and implementation of MPEG2Event. The second of these challenges will be addressed by our envisioned web service "MPEGstats" which will provide encoding statistics about commercially encoded MPEG2 video data.

The impact of providing this information to the research community will be to:

- Provide a statistical basis for developing realistic models of compressed video data that incorporates knowledge about how bits are allocated to different kinds of coding elements (i.e., headers, motion vectors, DCT coefficients, etc.).
- Create a common corpus of compressed video statistics that in turn will help normalize evaluation of different experimental systems and protocols.
- Spark insight into how multimedia systems and protocols can exploit the structure of encoded video data.
- Develop a large dataset of statistics which can then be mined for patterns and trends which may point toward larger scale structure and redundancy that exists with specific genres or subsets of related content.

4.2. Schema

In order to make the information deposited into MPEGstats uniform and complete, we needed to design a schema that specified exactly what information, measurements and statistics would be gleaned from each title and included in the MPEGstats database. We had two design goals for this schema. First, we wanted to include frame-level statistics that would help characterize each and every frame of the stream. Second, we wanted to keep the amount of data collected limited to a static well-known amount per frame in order make the size of the data collected a direct and linear function of the number of frames in the stream. We felt that this would facilitate resource planning when actually deploying the database and keep the size of the database to a reasonably manageable scale.

Our schema is comprised of 8 tables. The names and purpose of these tables are listed in Table 2. The bulk of the statistics exist in the "picture" table. Each entry in this table corresponds to the statistics gathered from one MPEG-2 encoded frame. Each entry is keyed to an entry in the "dvd" table. This table contains information about the title as a whole. Currently, only the title and number of chapters is captured. We can imagine extending this table to include information about genre, production date, etc.

Table 2: Schema table names and descriptions.

Table Name	Description
Dvd	DVD level information. Currently this table simply defines a unique ID which is used as an external key in the picture table and few basic pieces of information about the title.
Picture	Picture-level information. This table contains the bulk of the statistics gathered. Each entry in this table contains the statistics gathered from one frame of one title.
Coeff_avg	Coefficient averages. Each entry is a vector of 64 values with the average coefficient value by position for a particular frame. The entry is keyed to an entry in the picture table.
Coeff_bits	Number of bits used to encode each coefficient position for a particular picture.
Coeff_freq	Number of times each coefficient position was encoded within a particular picture.
Coeff_max	The maximum coefficient value for each coefficient position in a particular picture.
Coeff_min	The minimum coefficient value for each coefficient position in a particular picture.
Coeff_order	The number of different values encoded for each coefficient position in a particular picture.

For each picture, statistics are gathered about a number of different aspects about the encoding of that picture including: the encoded time code associated with the picture, the picture number relative to the beginning of the chapter, length in bytes, encoding type (i.e., I, P, or B), several different statistics about the value of the quantization scale, the number of macroblocks encoded in a particular way (i.e., intracoded, forward motion compensated, backward motion compensated, etc.), and a number of different statistics concerning the encoding and value of motion vectors. Additionally, aggregate statistics are gathered about the DCT coefficients encoded as part of that picture. These statistics are collected on a positional basis. In other words, separate statistics are gathered based on the coefficients position in the 2D coefficient array. We believe that this is important because the distribution of coefficients by position is highly non-uniform. Furthermore, higher-order coefficients (i.e., those that represent higher frequencies) are more heavily quantized and in general are considered less perceptually important. These statistics include: average coefficient value, number of bits used to encode the coefficients at this position, how often a coefficient in this position exists, the maximum and minimum values encoded at this position, and the number of different value encoded at this position. Because these statistics are gathered on a per position basis, each of these measures results in a 64 item vector since there are 64 different coefficient positions in the 2D 8x8 DCT used by MPEG-2. Each of these values is stored in a separate table (Coeff_avg, Coeff_bits, Coeff_freq, Coeff_max, Coeff_min, Coeff_order) and a GUID is assigned to act as a key for connecting the vector to the corresponding record in the picture table.

The design of the schema is a tradeoff between detail and space. In its current form, each frame of MPEG-2 video will generate approximately 200 bytes of statistics. At normal frame rates, this yields a data rate of 6kbs, or about 5MB per two hour movie. One weakness of the schema is the fact that the coefficient statistics are gathered relative to their encoded values as opposed to their decoded values. Because quantization is also positional, the same encoded coefficient value in two different coefficient positions may result in different decoded coefficient values. Another potential weakness is the fact that coefficient

information is not separated into those used to encode differential error as when motion compensation is in place from those used to directly encode pixel values. The picture type (i.e., I, P, or B), however, serves as a heuristic to make this classification since no motion compensation is used in I frames and the number of intracoded macroblocks in P or B frames is usually small. It should further be noted that this schema is only our initial design and that we plan on revisiting this design after gathering feedback from early users.

4.3. Interface

The interface to the MPEGstats information will be two-fold. First, we will develop a web-based interface through which a subset of the statistics can be accessed (Figure 9, Figure 10). A prototype of this interface is in place at http://www.mpegstats.msreasearch.us. This will serve to make the information immediately available to a wide community of users. Second, we will develop a web service interface to the backend database such that more complicated queries can be issued. The existence of integrated database connectivity in the .NET framework and well-defined XML-based web service interfaces such as SOAP make this task almost effortless.

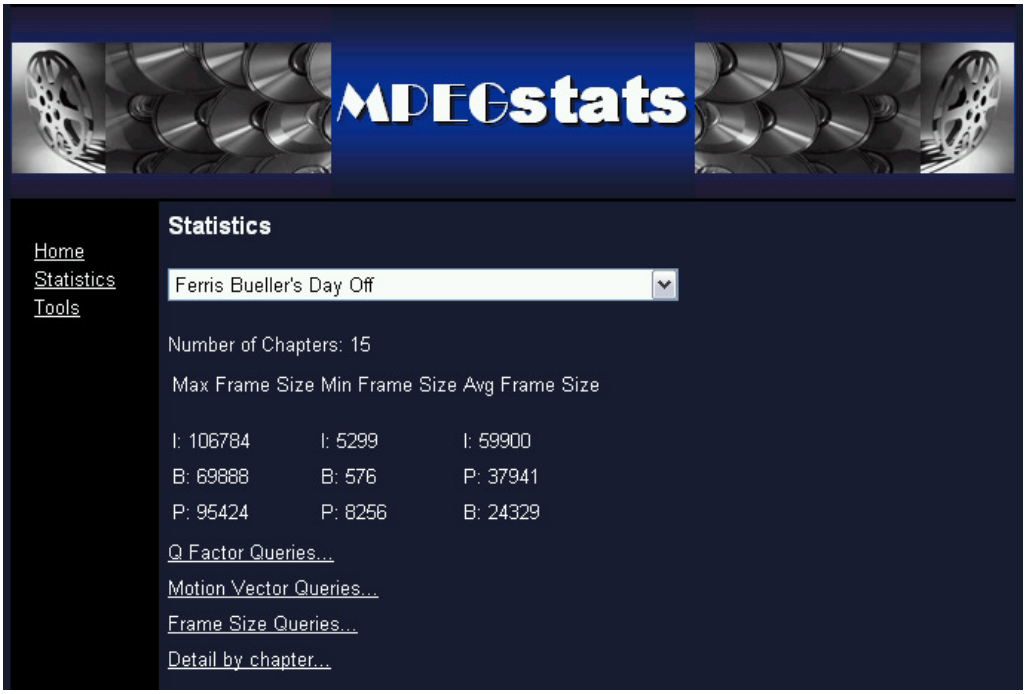
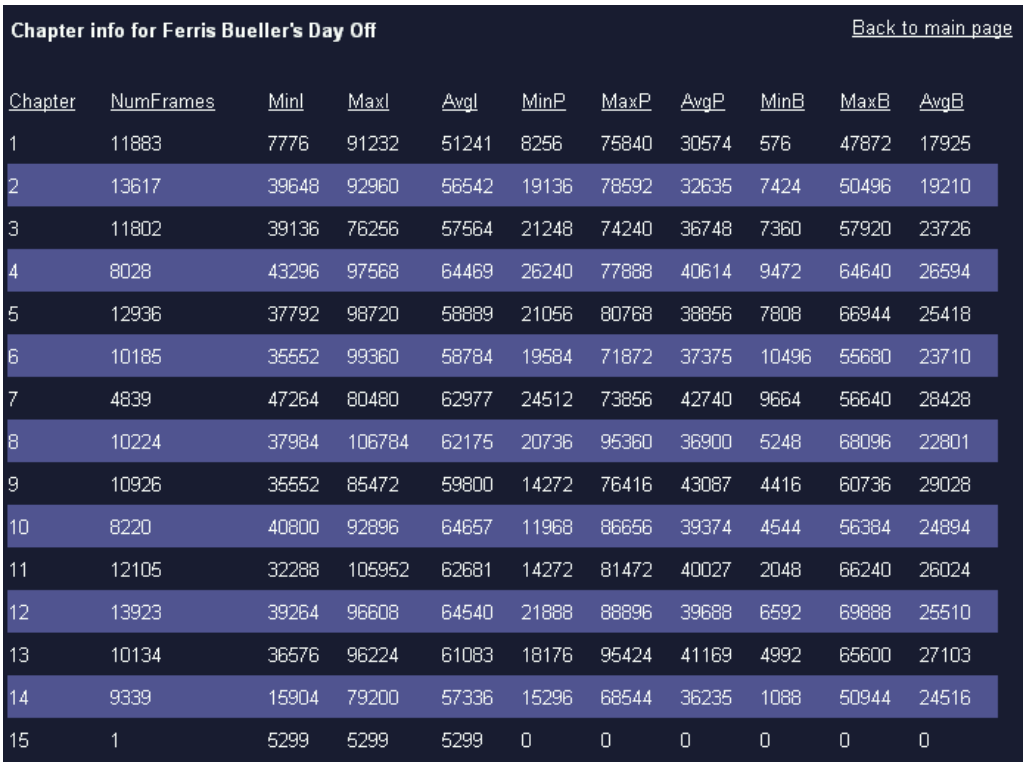


Figure 9 – MPEGstats web interface showing the number of chapters for a title along with the min, max and average frame sizes for I, B and P frames. Links for detailed queries are at the bottom.



Chapter info for Ferris Bueller's Day Off — Back to main page

Chapter	NumFrames	MinI	MaxI	AvgI	MinP	MaxP	AvgP	MinB	MaxB	AvgB
1	11883	7776	91232	51241	8256	75840	30574	576	47872	17925
2	13617	39648	92960	56542	19136	78592	32635	7424	50496	19210
3	11802	39136	76256	57564	21248	74240	36748	7360	57920	23726
4	8028	43296	97568	64469	26240	77888	40614	9472	64640	26594
5	12936	37792	98720	58889	21056	80768	38856	7808	66944	25418
6	10185	35552	99360	58784	19584	71872	37375	10496	55680	23710
7	4839	47264	80480	62977	24512	73856	42740	9664	56640	28428
8	10224	37984	106784	62175	20736	95360	36900	5248	68096	22801
9	10926	35552	85472	59800	14272	76416	43087	4416	60736	29028
10	8220	40800	92896	64657	11968	86656	39374	4544	56384	24894
11	12105	32288	105952	62681	14272	81472	40027	2048	66240	26024
12	13923	39264	96608	64540	21888	88896	39688	6592	69888	25510
13	10134	36576	96224	61083	18176	95424	41169	4992	65600	27103
14	9339	15904	79200	57336	15296	68544	36235	1088	50944	24516
15	1	5299	5299	5299	0	0	0	0	0	0

Figure 10 – MPEGstats web site; results from "detail by chapter" for a title.

4.4. Legality issues

We want to be careful not to violate any copyright laws in hosting this web site and web service. The publication of data on MPEGstats involves several steps:

1. Obtaining a decrypted MPEG stream from a DVD
2. Extracting the statistics via a program that uses the MPEG2Event library
3. Entering the statistics in the MPEGstats database
4. Making the statistics from the database available to the public via a web site and web service.

It is clear that steps 3 and 4 do not infringe any copyrights, as they involve only statistics, not the actual copyright material. In step 2, we note that at no time have we actually decoded the data and/or produced a visual picture either in memory or on the screen. Thus, we have only used the bits on the DVD as data without having viewed or having allowed to be viewed the visual information protected by the copyright.

The only potentially troubling step is step 1, in which, if an unauthorized decryption method is used, it might be construed as violating the DMCA. While we do not think there is actually a violation, since there is never any visualization of the result, nor any copies made, we will err on the side of caution by strictly separating steps 1/2 from steps 3/4. The web site operation will only publish material contributed by third parties.

5. Example: Exploring MPEGstats

In this section, we explain how we used MPEG2Event to answer a question about encoded video. The fundamental operation of data compression is the removal of redundancy. While completely general methods exist, within the specific domain of video compression, coding schemes rely on the highly structured nature of the data type. In particular, a pixel's color value is likely to be similar to those around it and the overall visual content is little changed from one frame to the next. Thus, frequency-domain representations (e.g., discrete cosine transform, wavelets, etc.) and interframe compression techniques (e.g., motion compensation, difference coding, conditional replenishment, etc.) are the mainstays of most coding

schemes.

These techniques are for the most part localized to a region within a relatively short sequence of frames. A longer-term, more global redundancy may be escaping the attention of current techniques. Intuitively, we experience this long-term redundancy as the "look-and-feel" of a particular movie or television show. We are able to almost instantly recognize our favorite shows even though we may not have seen the particular episode in question. The reason for this is because the visual characteristics of one episode (i.e., character faces, locations, sets, tone, lighting, etc.) are generally very much like any other episode. In the same way, the end of a movie may generally look like the beginning of a movie.

We asked the question: is it possible to employ this long-term redundancy to improve the compression rate and/or the robustness of a video stream? As a necessary preliminary, we first need to identify and quantify sources of long-term redundancy within existing video representations. To do this we wanted to analyze the coding elements of a particular video representation and try to answer basic questions such as:

- Do patterns within the coding elements exist that might be exploited?
- Do the distributions of certain coding element values match the distributions used by the standards to set variable bit length codes?
- What percentage of the bits in a video stream go to different functions such as meta information, motion compensation, error coding, frequency domain pixel coding, etc.?

By exploring the structure of existing video representations, we can answer these questions. We chose to concentrate on the MPEG-2 video standard because of its wide use in digital cable and satellite systems, DVD's, and personal video recorders such as TiVO. Furthermore, because MPEG is an open standard, we are able to deconstruct these video streams to any level of detail desired.

To answer our questions, we used MPEG2Event to extract the statistics from 20 DVDs and posted them in the MPEGstats database. We then composed a query to sum the sizes of each frame type (I, P, B). Looking at the results, we found that, on average, only 22% of the video was composed of I-frames. Therefore, no matter how drastically a new encoding scheme were to shrink I-frames, the savings could never exceed 22%. Thus MPEG2Event and MPEGstats quickly showed that there is not enough space to be saved to justify work on long-term redundancy.

6. Conclusion & Future Work

We have designed and implemented the MPEG2Event C# library which can be used to parse and decode an MPEG-2 video stream. Unlike existing decoders, the library allows rapid prototyping of analytical tools and exposes the different syntactic structures of the compressed video representation at a number of different levels of detail. Using this library, we have developed a tool for extracting picture-level statistics that is then used to populate a backend SQLServer database. We also developed the MPEGstats web interface to this database and plan on exporting a web service interface for making complex queries against this database. By populating this database with information about video streams taken from DVD's, we will provide other researchers with the ability to develop more accurate models of compressed video sources as well as begin to answer some of our original questions. The MPEGstats web site is now operational with 20 titles at http://mpegstats.msresearch.us and open source for the MPEG2Event library is available at http://www.cs.unc.edu/~kmp/mpeg2event/index.html.

At this point in time, we see three specific areas for future work: adding titles to the MPEGstats database, extending the MPEG2Event framework, and building analysis tools to quantify hidden long-term redundancy.

First and foremost, we need to populate the MPEGstats database and evangelize its existence to other researchers. Towards this end, we plan on hiring an undergraduate student at the University of North Carolina to work under the direction of Prof. Mayer-Patel during the summer. Although some development effort will be spent creating a turn-key system for extracting the statistics, this work is mostly mechanical. Populating the database is an important first step, however, as we hope to attract the attention and efforts of other researchers in the community. If we can achieve a critical mass of interest in the project, we hope that the community at large will then contribute statistics.

Second, we would like to find developers interested in extending the MPEG2Event framework to include capabilities for parsing program and transport streams as well as other coding standards such as MPEG-4 and JPEG2000. The basic framework has been written in a manner that should make this kind of extension straightforward. We expect much of this type of development work to be motivated by the work of researchers with specific project interests that require the additional functionality. Another avenue for further development is by using the library as part of course projects and having students write extensions. The framework has already proven useful for teaching purposes and was used in a graduate-level multimedia course at the University of North Carolina during the Spring 2004 semester.

Finally, we would like to rekindle our efforts in exploiting long-term redundancy in video. The first step along this path would be to use the statistics in MPEGstats to try to find patterns that point toward coding elements with non-uniform encoded distributions across time within a title or across different titles with common content (i.e., episodes of the same show, movies of the same genre, etc.). We can then develop specific analysis tools using the MPEG2Event library to quantify that redundancy. In particular, we are interested in exploting large client-side memories to support the representation and streaming of video. If we can identify a subset of syntactic structures (i.e., higher-level coding elements such as entire blocks or macroblocks) that holistically occur more often in a particular set of video streams, we can begin to develop a dictionary-based approach for coding and streaming. Doing so allows us to exploit episodic redundancy as well as to develop streaming protocols more resilient to loss and with superior rate-distortion tradeoffs.

References

[1] Information Technology - Generic Coding of Moving Pictures and Associated Audio Information: Video (H.262), International Telecommunication Union (ITU), July 1995.