MPEG-2 Decoding in a Stream Programming Language

Matthew Drake, Hank Hoffmann, Rodric Rabbah, and Saman Amarasinghe
Massachusetts Institute of Technology
Computer Science and Artificial Intelligence Laboratory
{madrake, hank, rabbah, saman}@mit.edu

Abstract

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. This paper describes an implementation of the MPEG-2 decoder using the StreamIt programming language. StreamIt is an architecture-independent stream language that aims to improve programmer productivity, while concomitantly exposing the inherent parallelism and communication topology of the application. 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. StreamIt allows for modular application development, which increases code reuse, and reduces the complexity of the debugging process since stream components can be verified independently. This in turn leads to greater programmer productivity.

1. Introduction

Image compression of still and motion pictures plays an important role in Internet and multimedia applications, digital appliances such as HDTV, and handheld devices. The compression process decreases data storage requirements (important for embedded devices) and provides higher effective transmission rates (important for Internet enabled devices).

Current programming practices often require developers to implement compression algorithms in low level languages and arduously tune their code for performance. This methodology is not cost effective because architecture-specific code is not portable and multiple implementations of the same codec are necessary. The process is made more challenging by the continuous evolution of standards, which are driven by new innovations in a rapidly growing digital multimedia market.

A typical compression algorithm involves three types of operations: data representation, lossy compression, and lossless compression. These operations are semi-autonomous, exhibit data and pipeline parallelism, and easily fit into a sequence of distinct processing stages. As such, image and video compression is a good match for the streaming model of computation where data are transformed by a series of filters, usually organized in well structured topologies. Stream programming models afford certain advantages in terms of programmability, robustness, and achieving high performance.

This paper describes an implementation of the widely used MPEG-2 compression standard in StreamIt [40], a high-level architecture-independent language for streaming computations. Our goal is to deliver a unified development environment that captures all aspects of stream application development without sacrificing either performance or programmability. This paper details the salient processing steps of the MPEG-2 decoder in StreamIt and compares some of the important implementation details to a reference C implementation of the decoder.

The StreamIt programming model allows the programmer to build an application by connecting components together into a stream graph, where the nodes represent filters that transform the data communicated along the edges. In StreamIt, the programmer is relieved of the burden of explicit buffer management and complex index expressions for multi-dimensional data. StreamIt also exposes the inherent parallelism and communication topology of the application, thereby empowering the compiler to perform many stream-aware optimizations [1, 17, 25, 37] that elude other languages. The end result is a clean, malleable, portable, and efficient code.

Our StreamIt development environment, optimizing compiler, and MPEG-2 codec implementations are available for download from our project webpage at http://cag.csail.mit.edu/streamit.

2. Overview of MPEG-2

MPEG-2 [19] is a popular encoding and decoding standard for digital video. The encoding process relies on *lossy* and *lossless* compression. Lossy compression permanently eliminates information from a video based on a human perception model. Humans are much better at discerning changes in color intensity (luminance information) than changes in hue (chrominance information). Humans are also more sensitive to low frequency image components, such as a blue sky, than to high frequency image components, such as a plaid shirt. Details which humans are likely to miss can be thrown away without affecting the perceived video quality.

The encoder operates on a sequence of pictures. Each picture is made from 16x16 groups of pixels known as macroblocks. A macroblock is composed from a series of blocks, which are 8x8 arrays of subpixels (individual color components of a pixel). The luminance information for a macroblock is a 2x2 array of blocks, whereas the chrominance channels are downsampled because of human insensitivity to hue. The type of downsampling in an MPEG-2 stream is known as its chroma format. The most common chroma format is 4:2:0. It uses one block for each chrominance channel, downsampling a macroblock from 16x16 to 8x8 subpixels. An alternate format is 4:2:2. It uses two blocks for each chrominance channel, downsampling each macroblock from 16x16 to 8x16 subpixels.

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. The result is a residual predictive-coded (P) picture. It represents the difference between the current picture and the reference picture. Reference pictures encoded without the use of motion prediction are intra-coded (I) pictures. In addition to forward motion prediction, it is possible to encode new pictures using motion estimation from both previous and subsequent pictures. Such pictures are bidirectionally predictive-coded (B) pictures, and they exploit a greater amount of temporal locality.

Each of the I, P, and B pictures then undergoes a 2-dimensional discrete cosine transform (DCT) which separates the picture into parts with varying visual importance. The input to the DCT is one block. The output of the DCT is an 8x8 matrix of frequency coefficients. The upper left corner of the matrix represents low frequencies, whereas the lower right corner represents higher frequencies. The latter are often small and can be neglected without sacrificing visual perception. The DCT block coefficients are quantized to reduce the number of bits needed to represent them.

Following quantization, many coefficients are effectively reduced to zero. The DCT matrix is stored in a run-length encoded format by emitting each non-zero coefficient, the number of bits needed to represent that coefficient, and the number of zero coefficients since the last non-zero coefficient. The run-length encoder scans the DCT matrix in a zig-zag order to consolidate the zeros in the matrix.

The output of the run-length encoder, motion vectors, picture type, and other picture and macroblock metadata are Huffman coded to further reduce the average number of bits per data item. The compressed stream is sent to the output device.

The decoder input stream is organized as a Group of Pictures (GOP) which contains all the information needed to reconstruct a video. The GOP contains the three kinds of pictures produced by the encoder, namely I, P, and B pictures. I pictures assist in scene cuts, random access, fast forwarding, or fast reverse playback [19]. A typical I:P:B picture ratio in a GOP is 1:3:8, and a typical picture pattern is a repetition of the following logical sequence: I_1 B_2 B_3 P_4 B_5 B_6 P_7 B_8 B_9 P_{10} B_{11} B_{12} I_{13} where the subscripts denote positions in the original video. However, to simplify the decoder, the encoder reorders the pictures to produce the following pattern: I_1 P_4 B_2 B_3 P_7 B_5 B_6 P_{10} B_8 B_9 I_{13} B_{11} B_{12}. Under this configuration, if the decoder encounters a P picture, its motion prediction is with respect to the previously decoded I or P picture; if the decoder encounters a B picture, its motion prediction is with respect to the previously two decoded I or P pictures.

The decoding process is conceptually the reverse of the encoding process. 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. Finally, the motion vectors parsed from the data stream are passed to a motion compensator, which reconstructs the original pictures.

```
int->int filter ZigZag(int N, int[N] Order) {
    work pop N push N {
        for (int i = 0; i < N; i++) {
            int pixel = peek(Order[i]);
            push(pixel);
        }
        for (int i = 0; i < N; i++) {
            pop();
        }
    }
}

int[64] Order =
        {00, 01, 05, 06, 14, 15, 27, 28,
         02, 04, 07, 13, 16, 26, 29, 42,
         03, 08, 12, 17, 25, 30, 41, 43,
         09, 11, 18, 24, 31, 40, 44, 53,
         10, 19, 23, 32, 39, 45, 52, 54,
         20, 22, 33, 38, 46, 51, 55, 60,
         21, 34, 37, 47, 50, 56, 59, 61,
         35, 36, 48, 49, 57, 58, 62, 63};
```

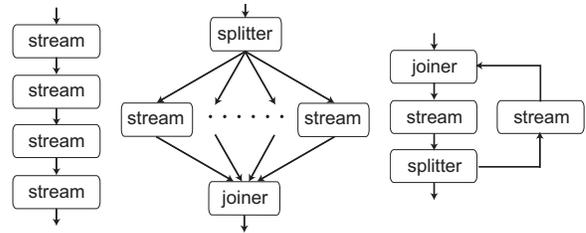**Figure 1. Zig-zag descrambling filter.**

3. StreamIt Programming Language

StreamIt [40] is an architecture independent language that is designed for stream programming. In StreamIt, programs are represented as graphs where nodes represent computation and edges represent FIFO-ordered communication of data over tapes. The language features several novelties that are essential for large scale program development. The language is modular, parameterizable, malleable and architecture independent. In addition, the language exposes the inherent parallelism and communication patterns that are prevalent in streaming programs.

3.1. Filters as Programmable Units

In StreamIt, the basic programmable unit is a *filter*. Each filter has an independent address space. Thus, all communication with other filters is via the input and output channels, and occasionally via control messages (see Section 3.3). The main filter method is the work function which represents a steady-state execution step. The work function pops (i.e., reads) items from the filter input tape and pushes (i.e., writes) items to the filter output tape. A filter may also peek at a given index on its input tape without consuming the item; this makes it simple to represent computation over a sliding window or to perform permutations on the input stream. The `push`, `pop`, and `peek` rates are declared as part of the work function, thereby enabling the compiler to apply various optimizations and construct efficient execution schedules.

A filter is parameterizable, and this allows for greater malleability and code reuse. An example filter is shown in Figure 1. This filter consumes a stream whose elements are of type `int` and produces a stream

(a) pipeline   (b) splitjoin   (c) feedback loop

Figure 2. Hierarchical streams in StreamIt.

of the same type. It implements the zig-zag descrambling 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. Each instantiation of a filter specifies the matrix dimensions, as well as the desired ordering. In MPEG, there are two possible scan orders. The `Order` parameter defines the specific scan pattern that is desired, as shown in Figure 1.

In this example, the input matrix is represented as a unidimensional stream of elements. The filter peeks the elements and copies them to the output stream in the specified order. Once all the DCT coefficients are copied, the input stream is deallocated from the tape with a series of pops. It has been shown that vector permutation instructions can be automatically generated from this representation [31].

3.2. Hierarchical Streams

In StreamIt, the application developer focuses on the hierarchical assembly of the stream graph and its communication topology, rather than the explicit management of the data buffers between filters. StreamIt provides three hierarchical structures for composing filters into larger stream graphs (see Figure 2).

Pipeline. A pipeline is a single input to single output parameterized stream. It composes streams in sequence, with the output of one connected to the input of the next. An example of a pipeline appears in Figure 3.

The `add` keyword in StreamIt constructs the specified stream using the input arguments. The `add` statement may only appear in non-filter streams. In essence, filters are the leaves in the hierarchical construction, and composite nodes in the stream graph define the encapsulating containers. This allows for modular design and development of large applications, thereby promoting collaboration, increasing code reuse, and simplifying debugging.

```
float->float pipeline IDCT_2D(int N) {
  // perform N 1D-IDCTs in parallel in the X direction
  add splitjoin {
    split roundrobin(N);
    for (int i = 0; i < N; i++)
      add IDCT_1D(N);
    join roundrobin(N);
  }
  // perform N 1D-IDCTs in parallel in the Y direction
  add splitjoin {
    split roundrobin(1);
    for (int i = 0; i < N; i++)
      add IDCT_1D(N);
    join roundrobin(1);
  }
}

float->float filter IDCT_1D(int N) {
  float[N][N] coeff = { ... };

  work pop N push N {
    for (int x = 0; x < N; x++) {
      float product = 0;
      for (int u = 0; u < N; u++)
        product += coeff[x][u] * peek(u);
      push(product);
    }
    for (int x = 0; x < N; x++) pop();
  }
}
```

Figure 3. Example pipeline and splitjoin.

```
// global variable
float coeff[64] = { ... };

void IDCT_2D(float* block) {
  int i, j, u;
  float product;
  float tmp[64];

  // 1D DCT in X direction
  for (i = 0; i < 8; i++)
    for (j = 0; j < 8; j++) {
      product = 0;

      for (u = 0; u < 8; u++)
        product += coeff[u][j] * block[8*i + u];

      tmp[8*i + j] = product;
    }

  // 1D DCT in Y direction
  for (j = 0; j < 8; j++)
    for (i = 0; i < 8; i++) {
      product = 0;

      for (u = 0; u < 8; u++)
        product += coeff[u][i] * tmp[8*u + j];

      block[8*i + j] = product;
    }
}
```

Figure 4. Example C code for 2D inverse DCT calculation using two 1D transforms.

Split-Join. The splitjoin construct distributes data to a set of parallel streams, which are then joined together in a roundrobin fashion. In a splitjoin, the *splitter* performs the data scattering, and the *joiner* performs the gathering. A splitter is a specialized filter with a single input and multiple output channels. On every execution step, it can distribute its output to any one of its children in either a *duplicate* or a *roundrobin* manner. A duplicate splitter (indicated by `split duplicate`) replicates incoming data to each stream connected to the splitter. A roundrobin splitter (indicated by `split roundrobin(`$w_1, \ldots, w_n$`)`) distributes the first w_1 items to the first child, the next w_2 items to the second child, and so on. The splitter counterpart is the joiner. It gathers data from its predecessors in a roundrobin manner to produce a single output stream.

The splitjoin and pipeline constructs provide a convenient and natural way to represent parallel computation. An example is shown in Figure 3, which illustrates a parallel implementation of the 2D inverse DCT using 1D inverse DCTs. This implementation is both data parallel (within the rows and columns) and pipeline parallel (between the rows and columns). A straightforward C implementation of a computationally equivalent inverse DCT is shown in Figure 4. Note that the code structure is similar to the StreamIt version, although it does not explicitly expose the parallelism. The C code also requires explicit array index management, such as the expressions `block[8*i+u]` and `tmp[8*i+j]` which are notably absent in the StreamIt code. The splitter and joiner in StreamIt free the programmer from tedious indexing operations, which also enables the compiler to understand and optimize the buffer management [37]. The StreamIt implementation is also parameterized such that it is trivial to adjust the size of the inverse DCT.

Feedback Loop. StreamIt also provides a feedback loop construct for introducing cycles in the graph. This stream construct is not used in the decoder, but is useful in other applications.

3.3. Teleport Messaging

A notoriously difficult aspect of stream programming, from both a performance and programmability standpoint, is reconciling regular streaming dataflow with irregular control messages. While the high-bandwidth flow of data is very predictable, realistic applications such as MPEG also include unpredictable, low-bandwidth control messages for adjusting system parameters (e.g., desired precision in quantization, type of picture, resolution, etc.).

For example, the inverse quantization step in the decoder uses a lookup table that provides the inverse quantization scaling factors. However, the particular scaling factor is determined by the stream parser.

Since the parsing and inverse quantization tasks are logically decoupled, any pertinent information that the parser discovers must be forwarded to the appropriate streams. In StreamIt, such communication is conveniently accomplished using teleport messaging [41].

The idea behind teleport messaging is for the `Parser` to change the quantization precision via an asynchronous method call, where method invocations in the target are timed relative to the flow of data in the stream (i.e., macroblocks). As shown in Figure 5, the `InverseDCQuantizer` declares a message handler that adjusts its precision (lines 27-29). The `Parser` calls this handler through a *portal* (line 16), which provides a clean interface for messaging. The handler invocation includes a range of latencies `[min:max]` specifying when the message should be delivered with respect to the data produced by the sender.

Intuitively, the message semantics can be understood as tags attached to data items. If the `Parser` sends a message to a filter downstream (i.e., in the same direction as dataflow) with a latency k, then, conceptually, the filter tags the items that it outputs in k iterations of its work function. If $k = 0$, the data produced in the current execution of the work function is tagged. The tags propagate through the stream graph; whenever a filter inputs an item that is tagged, all of its subsequent outputs are also tagged. The message flows through the graph until the first tagged data item reaches the intended receiver, at which time the message handler is invoked immediately before the execution of the work function in the receiver. In this sense, the message has the semantics of traveling "with the data" through the stream graph, even though it is not necessarily implemented this way.

Teleport messaging avoids the muddling of data streams with control-relevant information. Teleport messaging thus separates the concerns of the programmer from the system implementation, thereby allowing the compiler to deliver the message in the most efficient way for a given architecture. In addition, by exposing the exact data dependences to the compiler, filter executions can be reordered so long as they respect the message timing. Such reordering is generally impossible if control information is passed via global variables. Teleport messaging also offers other powerful control over timing and latency beyond what is used in this example (in particular, the ability to send messages opposite the direction of dataflow [41]).

4. MPEG Decoder in StreamIt

The MPEG decoder pipeline is shown in Figure 6. The stream graph is shown on the left. The StreamIt

```
01 void->void MPEGDecoder {
02   ...
03   portal<InverseDCQuantizer> p;
04   ...
05   add Parser(p);
06   ...
07   add InverseDCQuantizer() to p;
08   ...
09 }

10 int->int filter Parser(portal<InverseDCQuantizer> p) {
11   work push * {
12     int precision;
13     ...
14     if (...) {
15       precision = pop();
16       p.setPrecision(precision) [0:0];
17     }
18     ...
19   }
20 }

21 int->int filter InverseDCQuantizer() {
22   int[4] scalingFactor = {8, 4, 2, 1};
23   int    precision = 0;

24   work pop 1 push 1 {
25     push(scalingFactor[precision] * pop());
26   }

27   handler setPrecision(int new_precision) {
28     precision = new_precision;
29   }
30 }
```

Figure 5. Messaging example.

code is shown on the right, and it is correlated with the stream block level diagram.

The computation is encapsulated in three main components: the parser (line 8), the block and motion vector decoder (lines 9-22), and the motion compensator (lines 23-32). The parser is responsible for parsing the MPEG-2 bit stream and performing Huffman and variable run-length decoding (VLD). The output of the VLD is an interleaved stream of quantized macroblocks encoded in the frequency-domain, and offset-encoded motion vectors. The VLD outputs $N \times B$ data elements for each macroblock, followed by V data elements that encode its motion vector. The actual value of N depends on the chroma format. In a 4:2:0 chroma format, $N = 6$ since each macroblock consists of four 8x8 subpixel blocks for the luminance channel, and two 8x8 subpixel blocks for the two chrominance channels. Therefore, the VLD outputs a total of six 8x8 blocks, or 384 subpixels per macroblock. The I/O rate of the parser varies with the compression ratio of the input stream. The VLD filter is the only variable rate filter in the decoder pipeline.

The VLD output is segregated into two homogeneous streams by a roundrobin splitter (line 10). The first stream undergoes inverse transformations (lines 11-16), while the second is decoded to produce absolute motion vectors (lines 17-20). As is evident from the computation graph, the two streams are decoded in

parallel, and then merged (line 21) prior to the motion compensation stage of the pipeline.

The inverse transformations map each 8x8 block from the frequency domain back to the spatial domain. Each block is reordered (line 12), and then inversely quantized (line 13). This is followed by an inverse DCT and a bounded saturation filter (lines 14-15). The set of transformations is grouped into a pipeline whose input and output types are automatically inferred by the compiler. Each of the filters in this pipeline operates on 8x8 blocks. The code that is shown does not take advantage of data level parallelism between blocks. It is rather straightforward however to expose this parallelism if it is desirable. For example, in this case a splitjoin can replicate the inverse transformation pipeline N times:

```
add splitjoin {
  split roundrobin(B);
  for (int i = 0; i < N; i++)
    // add pipeline
  join roundrobin(B);
}
```

A stream-aware compiler can also automatically adjust the execution granularity as necessary [17], since data-parallel streams can be easily identified as those that are stateless (i.e., do not carry mutable state from one iteration to the next).

The third stage of the decoding pipeline performs the motion compensation (lines 23-32) to recover predictively coded macroblocks. 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, there are three parallel streams. The first handles the luminance color channel (Y), and the other two handle the chrominance channels (Cb and Cr). The roundrobin splitter (line 24) distributes the macroblocks according to the chroma format. Since the luminance channel is not downsampled during the encoding process, the splitter dispatches four 8x8 blocks at a time to the Y motion compensator. The chrominance channels are typically downsampled by a factor of 4, and hence one 8x8 block is streamed to each of the Cb and Cr pipelines, which upsample (line 29) the results of the motion compensator to generate 16x16 macroblocks. The upsampling is a linear interpolation of the surrounding pixels. The joiner (line 31) assembles the pictures from each of the color channels, one pixel at a time. The output is then readied for display (lines 33 and 34) by organizing the pictures in accord with their temporal order, and performing color space conversion to the RGB (red, green, blue)

color model. Note that these two filters each consume $3 \times W \times H$ subpixels per picture. This is three times the resolution of the decoded image since there is one pixel generated from each of the three channel decoders. The final output of the decoder is $W \times H$ pixels. In contrast to the filters for motion compensation and inverse transformation, whose I/O rates are statically resolved at compile time, the picture reordering and color space conversion have I/O rates that are parameterized on initialization time constants, namely the resolution of the pictures.

The decoder implementation was carried out by one student programmer with no prior understanding of MPEG. The development spanned eight weeks from specification [19] to the first fully functional MPEG decoder. The StreamIt code is nearly 3,165 lines of code with 48 static streams. The stream parser is the largest single filter, consisting of 775 lines of code. The 48 static streams compile to 2,150 instantiated filters[1] at a picture resolution of 352x240. By way of comparison, the reference C implementation [43] is 6,835 lines of code[2]. A line count comparison is not an accurate measure of programmability, since our StreamIt decoder implements only a subset of several stream types supported by MPEG. Our decoder does provide full support for the range of different compression techniques used within MPEG, but supports only a subset of the possible display modes (i.e., interlaced versus progressive output). However, these alternate display formats represent minor conceptual changes and should therefore affect small portions of the StreamIt code. This is demonstrated in Section 5 with an example that illustrates how to support multiple chrominance formats.

The reference C implementation intermingles parsing, decoding, and motion compensation, making it difficult to clearly follow the code, and hindering a better comparison. The C code also relies on global variables to communicate values, such as quantization coefficients, from the parser to the relevant code regions. In StreamIt, such communication is relegated to teleport messaging (lines 13, 25, 28, and 33, and illustrated with dotted lines in Figure 6). For instance, the parser (VLD) generates a message whenever the picture or macroblock type changes. The motion compensation filters receive this information via their dedicated portal (line 28), determine how to process the current picture, and decide whether they need to store the picture for future reference. Note however that while there are

---

[1]A *static stream* is a unique code block, which may have multiple instantiations. For instance, `MotionCompensation()` is a single filter with three instantiations.

[2]Line counts were generated using the `SLOCcount` tool. It strips whitespace and comments.

multiple motion compensators subscribed to the same portal, they each receive messages with respect to their local execution. The picture reordering filter receives a similar message (via portal on line 33), and uses the information to determine the correct temporal order of pictures. The inverse transformation pipeline listens to its portal (line 13) to determine the algebraic manipulation required to perform the inverse quantization of the input macroblock. Teleport messaging exposes the flow of messages to the compiler, and allows for large scale reordering or parallelization of the application without a heroic dependence analysis. It also provides a mechanism to easily introduce dynamic behavior into an otherwise static processing pipeline.

In StreamIt, all of the processing is encapsulated hierarchically into single-input, single-output streams with well-defined modular interfaces. This facilitates development and boosts programmer productivity, as components can be debugged and verified as standalone components. The modularity also promotes reuse. For example, the zig-zag descrambler and inverse DCT can be used as-is in a JPEG decoder.

5. Code Malleability: A Case Study

A noteworthy aspect of the StreamIt implementation is its malleability. We illustrate this by outlining how the decoder implementation is modified to support both 4:2:0 and 4:2:2 chroma formats. MPEG-2 streams are typically encoded using the former, which achieves a 50% reduction in the number of blocks required to represent a video. A higher sampling rate retains more color information from the original picture, and results in better overall quality.

The conceptual difference between chroma formats is merely a change in downsampling ratio. The change affects the data I/O rates, and the ratios of data between color channels. In the C reference code, the change requires adjustments to buffer sizes, array lengths, array indices, loop bounds, and various pointer offsets. The reference implementation uses a chroma
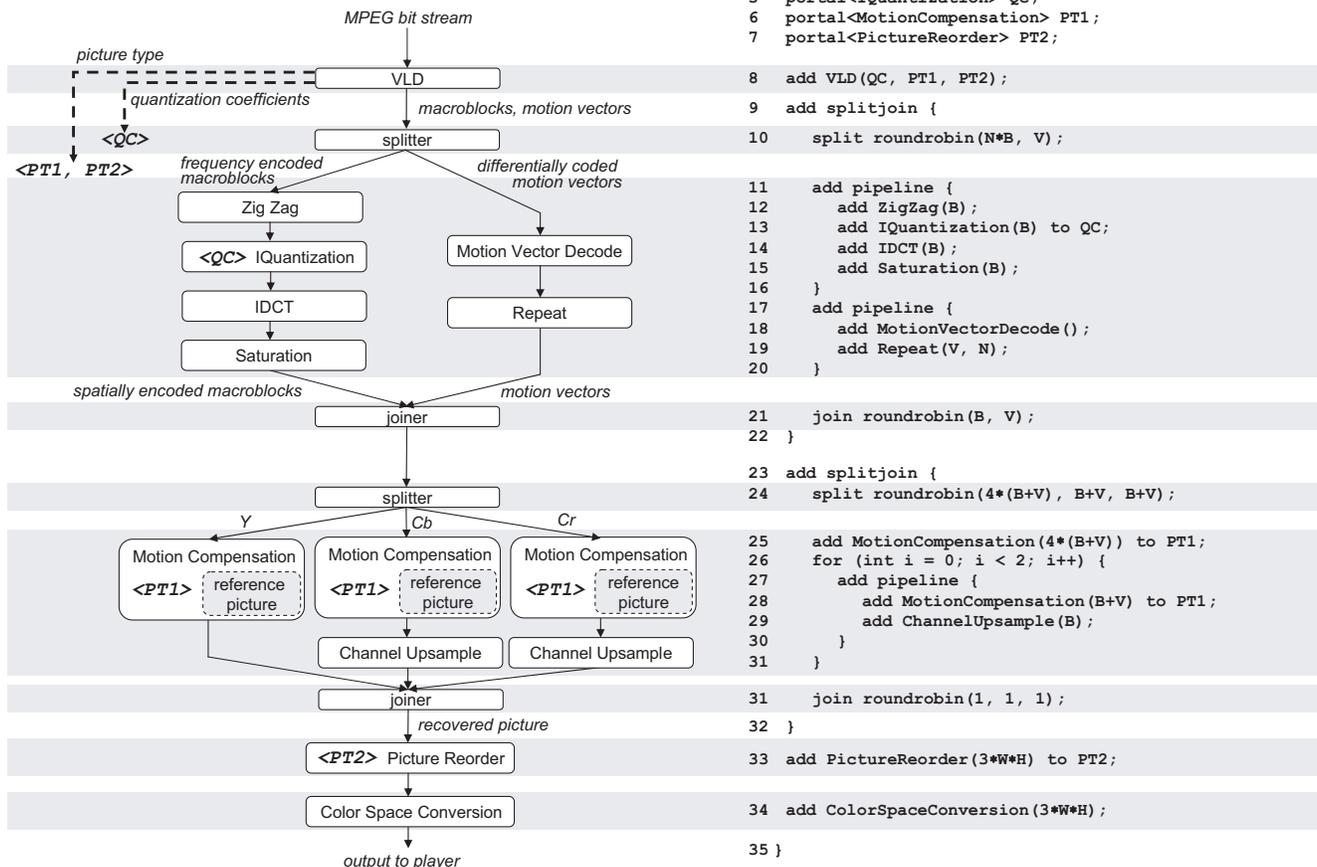
```
1 bit->int pipeline MPEG Decoder(int W, int H) {
     // W, H represent picture resolution in pixels
2    int B = 8*8; // block size in pixels
3    int V = 8;   // motion vector size
4    int N = 6;   // blocks per macroblock
                  // (4:2:0 format)

5    portal<IQuantization> QC;
6    portal<MotionCompensation> PT1;
7    portal<PictureReorder> PT2;

8    add VLD(QC, PT1, PT2);
9    add splitjoin {
10      split roundrobin(N*B, V);

11      add pipeline {
12         add ZigZag(B);
13         add IQuantization(B) to QC;
14         add IDCT(B);
15         add Saturation(B);
16      }
17      add pipeline {
18         add MotionVectorDecode();
19         add Repeat(V, N);
20      }

21      join roundrobin(B, V);
22   }

23   add splitjoin {
24      split roundrobin(4*(B+V), B+V, B+V);

25      add MotionCompensation(4*(B+V)) to PT1;
26      for (int i = 0; i < 2; i++) {
27         add pipeline {
28            add MotionCompensation(B+V) to PT1;
29            add ChannelUpsample(B);
30         }
31      }

31      join roundrobin(1, 1, 1);
32   }

33   add PictureReorder(3*W*H) to PT2;

34   add ColorSpaceConversion(3*W*H);

35 }
```


Figure 6. Block diagram of MPEG-2 decoder and corresponding StreamIt code.

```
01 /* Y */                                                    // C = blocks per chroma channel per macroblock
02 form_component_prediction(src[0]+(sfield?lx2>>1:0),dst[0]+(dfield?lx2>>1:0),    // C = 1 for 4:2:0, C = 2 for 4:2:2
03                      lx,lx2,w,h,x,y,dx,dy,average_flag);    add splitjoin {
04 if (chroma_format!=CHROMA444)  {                                split roundrobin(4*(B+V), 2*C*(B+V));
05    lx>>=1; lx2>>=1; w>>=1; x>>=1; dx/=2;                        add MotionCompensation() to PT1;
06 }                                                               add splitjoin {
07 if (chroma_format==CHROMA420) {                                    split roundrobin(B+V, B+V);
08    h>>=1; y>>=1; dy/=2;                                           for (int i = 0; i < 2; i++) {
09 }                                                                     add MotionCompensation(B+V) to PT1;
10 /* Cb */                                                             add ChannelUpsample(C*B);
11 form_component_prediction(src[1]+(sfield?lx2>>1:0),dst[1]+(dfield?lx2>>1:0),     }
12                      lx,lx2,w,h,x,y,dx,dy,average_flag);          join roundrobin(1, 1);
13 /* Cr */                                                        }
14 form_component_prediction(src[2]+(sfield?lx2>>1:0),dst[2]+(dfield?lx2>>1:0),  join roundrobin(1, 1, 1);
15                      lx,lx2,w,h,x,y,dx,dy,average_flag);    }
```

Figure 7. C (left) and StreamIt (right) code excerpts for handling 4:2:0 and 4:2:2 chroma formats.

flag to dictate control flow and alternate index/offset calculations in 43 locations in the code. As an example, Figure 7 shows a code fragment from the form_prediction routine in recon.c [43]. The function calls a subroutine to perform the motion compensation on each of the three color channels, passing in array offsets to a global array holding the data. Lines 4-6 adjust values used for address calculations to handle the 4:2:2 and 4:2:0 chroma formats, and lines 7-9 provide additional adjustments for the 4:2:0 format. While these offset adjustments are necessary in C, they are difficult for programmers and make the code hard to understand.

In StreamIt, we modified 31 lines and added 20 new lines to support the 4:2:2 format. Of the 31 modified lines, 23 were trivial changes to introduce the chroma format as a stream parameter. The greatest substantial change was to the color channel splitter, previously illustrated on line 24 of Figure 6. In the case of a 4:2:2 sampling rate, the chrominance data, as they appear on the input tape, alternate between each of the two chrominance channels. That is, the pattern of blocks is $Y_1Y_2Y_3Y_4Cb_5Cr_6Cb_7Cr_8$ (the pattern for the 4:2:0 case omits the last two blocks). Thus, a nested splitjoin is used to properly recover the chrominance channels. The new splitjoin is shown in the right half of Figure 7. In the StreamIt code, the chroma format explicitly dictates control flow in only 9 locations. Of course, the scheduling and buffer management changes dramatically between chroma formats, but this is transparent to the programmer.

6. Related Work

Video codecs have been a longtime focus of the embedded and high-performance computing communities. We consider related work in modeling environments, stream languages and parallel computing.

There have been many efforts to develop expressive and efficient models of computation for use in rapid prototyping environments such as Ptolemy [28],

GRAPE-II [26], and COSSAP [24]. The Synchronous Dataflow model (SDF) represents computation as an independent set of actors that communicate at fixed rates [27]. StreamIt leverages the SDF model of computation, though also supports dynamic communication rates and out-of-band control messages. There are other extensions to SDF that provide similar dynamic constructs. Synchronous Piggybacked Dataflow (SPDF) supports control messages in the form of a global state table with well-timed reads and writes [33, 34]. SPDF is evaluated using MP3 decoding, and would also be effective for MPEG-2 decoding. However, control messages in StreamIt are more expressive than SPDF, as they allow messages to travel upstream (opposite the direction of dataflow), with adjustable latency, and with more fine-grained delivery (i.e., allowing multiple execution phases per actor and multiple messages per phase). Moreover, our focus is on providing a high-level programming abstraction rather than an underlying model of computation.

Ko and Bhattacharyya also extend SDF with the dynamism needed for MPEG-2 encoding; they use "blocked dataflow" to reconfigure sub-graphs based on parameters embedded in the data stream [23] and a "dynamic graph topology" to extend compile-time scheduling optimizations to each runtime possibility [22]. Neuendorffer and Lee also extend SDF to support hierarchical parameter reconfiguration, subject to semantic constraints [32]. Unlike our description of control messages, these models allow reconfiguration of filter I/O rates and thus require alternate or parameterized schedules. MPEG-2 encoding has also been expressed in formalisms such as Petri nets [42] and process algebras [36].

There are a number of stream-oriented languages besides StreamIt, drawing from functional, dataflow, CSP and synchronous programming styles [39]. Synchronous languages which target embedded applications include Esterel [7], Lustre [18], Signal [16], Lucid [5], and Lucid Synchrone [9]. Additional languages of recent interest are Cg [30], Brook [8], Spi-

dle [10], StreamC/KernelC [21], Occam[11], Parallel Haskell [4] and Sisal [15]. The primary differences between StreamIt and these languages are (*i*) StreamIt supports (but is no longer limited to) the Synchronous Dataflow [27] model of computation, (*ii*) StreamIt offers a "peek" construct that inspects an item without consuming it from the channel, (*iii*) the single-input, single-output hierarchical structure that StreamIt imposes on the stream graph, and (*iv*) the teleport messaging feature for out-of-band communication.

Many researchers have developed both hardware and software schemes for parallel video compression; see Ahmad et al. [3] and Shen et al. [38] for reviews. We focus on programming models used to implement MPEG on general-purpose hardware. Assayad et al. present a syntax of parallel tasks, forall loops, and dependence annotations for exposing fine-grained parallelism in an MPEG-4 encoder [6]. A series of loop transformations (currently done by hand) lowers the representation to an MPI program for an SMP target. The system allows parallel components to communicate some values through shared memory, with execution constraints specified by the programmer. In comparison, StreamIt adopts a pure dataflow model with a focus on making the programming model as simple as possible. Another programming model is the Y-Chart Applications Programmers Interface (YAPI) [13], which is a C++ runtime library extending Kahn process networks with flexible channel selection. Researchers have used YAPI to leverage programmer-extracted parallelism in JPEG [12] and MPEG-2 [14]. Other high-performance programming models for MPEG-2 include manual conversion of C/C++ to SystemC [35], manual conversion to POSIX threads [29], and custom mappings to multiprocessors [2, 20]. Our focus again lies on the programmability: StreamIt provides an architecture-independent representation that is natural for the programmer while exposing pipeline and data parallelism to the compiler.

7. Concluding Remarks

In this paper we described our MPEG-2 codec implementation as it was developed using the StreamIt programming language. Our MPEG-2 decoder was developed in eight weeks by a single student programmer with no prior MPEG knowledge. We showed how the implementation is malleable by describing how the decoder is modified to support two different chrominance sampling rates. In addition, we showed that the StreamIt language is a good match for representing the MPEG stream flow because there is a direct correlation between the block level diagram describing the flow of data between computation elements and the

application syntax. Furthermore, we illustrated that teleport messaging, which allows for out-of-band communication of control parameters, allows the decoder to decouple the regular flow of data from the irregular communication of parameters (e.g., quantization coefficients). This in turns leads to a cleaner implementation that is easier to maintain and evolve with changing software specifications.

As computer architectures change from the traditional monolithic processors, to scalable wire-exposed and multi-core processors, there will be a greater need for portable codec implementations that expose parallelism and communication to enable efficient and high performance executions—while also boosting programmer productivity. StreamIt represents a step toward this end by providing a language that features hierarchical, modular, malleable, and portable streams.

Acknowledgements

References

[1] S. Agrawal, W. Thies, and S. Amarasinghe. Optimizing stream programs using linear state space analysis. In *CASES*, 2005.

[2] I. Ahmad, S. M. Akramullah, M. L. Liou, and M. Kafeel. A Scalable Off-line MPEG-2 Video Encoding Scheme using a Multiprocessor System. *Parallel Computing*, 27, 2001.

[3] I. Ahmad, Y. He, and M. L. Liou. Video compression with parallel processing. *Parallel Computing*, 28, 2002.

[4] S. Aidtya, Arvind, L. Augustsson, J. Maessen, and R. S. Nikhil. Semantics of pH: A parallel dialect of Haskell. In *Haskell Workshop*, 1995.

[5] E. Ashcroft and W. Wadge. Lucid, a non procedural language with iteration. *C. ACM*, 20(7), 1977.

[6] I. Assayad, P. Gerner, S. Yovine, and V. Bertin. Modelling, Analysis and Parallel Implementation of an Online Video Encoder. In *1st Int. Conf. on Distributed Frameworks for Multimedia Applications*, 2005.

[7] G. Berry and G. Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Sci. of Comp. Programming*, 19(2), 1992.

[8] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. In *SIGGRAPH*, 2004.

[9] P. Caspi and M. Pouzet. Lucid Synchrone distribution. http://www-spi.lip6.fr/lucid-synchrone/.

[10] C. Consel, H. Hamdi, L. Rveillre, L. Singaravelu, H. Yu, and C. Pu. Spidle: A DSL Approach to Specifying Streaming Application. In *2nd Int. Conf. on Generative Prog. and Component Engineering*, 2003.

[11] I. Corporation. *Occam 2 Reference Manual*. Prentice Hall, 1988.

[12] E. de Kock. Multiprocessor Mapping of Process Networks: A JPEG Decoding Case Study. In *15th Int. Symp. on System Synthesis*, 2002.

[13] E. de Kock, G. Essink, W. Smits, P. van der Wolf, J. Brunel, W. Kruijtzer, P. Lieverse, and K. Vissers. YAPI: Application Modeling for Signal Processing Systems. In *Conf. on Design Automation*, 2000.

[14] B. K. Dwivedi, J. Hoogerbrugge, P. Stravers, and M. Balakrishnan. Exploring design space of parallel realizations: MPEG-2 decoder case study. In *9th Int. Symp. on Hardware/Software Codesign*, 2001.

[15] J. Gaudiot, W. Bohm, T. DeBoni, J. Feo, and P. Mille. The Sisal Model of Functional Programming and its Implementation. In *2nd Aizu Int. Symposium on Parallel Algorithms/Architecture Synthesis*, 1997.

[16] T. Gautier, P. L. Guernic, and L. Besnard. Signal: A declarative language for synchronous programming of real-time systems. *Springer Verlag LNCS*, 274, 1987.

[17] M. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, C. Leger, A. A. Lamb, J. Wong, H. Hoffman, D. Z. Maze, and S. Amarasinghe. A Stream Compiler for Communication-Exposed Architectures. In *ASPLOS*, 2002.

[18] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow language LUSTRE. *Proc. of the IEEE*, 79(1), 1991.

[19] ISO/IEC 11172: Information technology — Coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s. International Organization for Standardization, 1999.

[20] E. Iwata and K. Olukotun. Exploiting coarse-grain parallelism in the MPEG-2 algorithm. Technical Report CSL-TR-98-771, Stanford University, 1998.

[21] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens. Programmable stream processors. *IEEE Computer*, 2003.

[22] D.-I. Ko and S. S. Bhattacharyya. Dynamic Configuration of Dataflow Graph Topology for DSP System Design. In *ICASSP*, 2005.

[23] D.-I. Ko and S. S. Bhattacharyya. Modeling of Block-Based DSP Systems. *Journal of VLSI Signal Processing*, 40(3), 2005.

[24] J. Kunkel. COSSAP: A stream driven simulator. In *Int. Workshop on Microelectronics in Communications*, 1991.

[25] A. A. Lamb, W. Thies, and S. Amarasinghe. Linear Analysis and Optimization of Stream Programs. In *PLDI*, 2003.

[26] R. Lauwereins, M. Engels, M. Adé, and J. Peperstraete. Grape-II: A System-Level Prototyping Environment for DSP Applications. *IEEE Computer*, 28(2), 1995.

[27] E. Lee and D. Messershmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Trans. on Computers*, C-36(1), 1987.

[28] E. A. Lee. Overview of the Ptolemy Project. Technical report, UCB/ERL M03/25, UC Berkeley, 2003.

[29] M.-L. Li, R. Sasanka, S. V. Adve, Y.-K. Chen, and E. Debes. The ALPBench Benchmark Suite for Complex Multimedia Applications. In *IEEE Int. Symp. on Workload Characterization*, 2005.

[30] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: A System for Programming Graphics Hardware in a C-like Language. In *SIGGRAPH*, 2003.

[31] M. Narayanan and K. A. Yelick. Generating permutation instructions from a high-level description. In *Workshop on Media and Streaming Processors*, 2004.

[32] S. Neuendorffer and E. Lee. Hierarchical Reconfiguration of Dataflow Models. In *Conference on Formal Methods and Models for Codesign*, 2004.

[33] C. Park, J. Chung, and S. Ha. Efficient Dataflow Representation of MPEG-1 Audio (Layer III) Decoder Algorithm with Controlled Global States. In *IEEE Workshop on Signal Processing Systems*, 1999.

[34] C. Park, J. Jung, and S. Ha. Extended Synchronous Dataflow for Efficient DSP System Prototyping. *Design Automation for Embedded Systems*, 6(3), 2002.

[35] N. Pazos, P. Ienne, Y. Leblebici, and A. Maxiaguine. Parallel Modelling Paradigm in Multimedia Applications: Mapping and Scheduling onto a Multi-Processor System-on-Chip Platform. In *Int. Global Signal Processing Conference*, 2004.

[36] F. L. Pelayo, F. Cuartero, V. Valero, D. Cazorla, and T. Olivares. Specification and Performance of the MPEG-2 Video Encoder by Using the Stochastic Process Algebra: ROSA. In *17th UK Performance Evaluation Workshop*, 2001.

[37] J. Sermulins, W. Thies, R. Rabbah, and S. Amarasinghe. Cache Aware Optimization of Stream Programs. In *LCTES*, 2005.

[38] K. Shen, G. Cook, L. Jamieson, and E. Delp. Overview of parallel processing approaches to image and video compression. In *SPIE Conference on Image and Video Compression*, 1994.

[39] R. Stephens. A Survey of Stream Processing. *Acta Informatica*, 34(7), 1997.

[40] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Int. Conf. on Compiler Construction*, 2002.

[41] W. Thies, M. Karczmarek, J. Sermulins, R. Rabbah, and S. Amarasinghe. Teleport messaging for distributed stream programs. In *PPoPP*, 2005.

[42] V. Valero, F. L. Pelayo, F. Cuartero, and D. Cazorla. Specification and Analysis of the MPEG-2 Video Encoder with Timed-Arc Petri Nets. *Electronic Notes in Theoretical Computer Science*, 66(2), 2002.

[43] VMPEG (Reference C Code). ftp://ftp.mpegtv.com/pub/mpeg/mssg/mpeg2vidcodec_v12.tar.gz.