MPEG-2 in a Stream Programming Language

Matthew Drake, Hank Hoffmann, Rodric Rabbah, and Saman Amarasinghe
MIT Computer Science and Artificial Intelligence Laboratory

Abstract

Image and video codecs are prevalent in multimedia applications, ranging from embedded systems, to desktop computers, to high-end servers such as HDTV editing consoles. It is not uncommon however that developers create (from scratch) and customize their codec implementations for each of the architecture targets they intend their coders and decoders to run on. This practice is time consuming and error prone, leading to code that is not malleable or portable. In this paper we describe an implementation of the MPEG-2 codec 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. We describe why MPEG is a good match for the streaming programming model, and illustrate 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 also reduces the complexity of the debugging process since stream components can be verified independently. This in turn leads to greater programmer productivity. We implement a fully functional MPEG-2 decoder in StreamIt. The decoder was developed in eight weeks by a single student programmer who did not have any prior experience with MPEG or other video codecs. Many of the MPEG-2 components were subsequently reused to assemble a JPEG codec.

1. Introduction

Image compression, whether for still pictures or motion pictures (e.g., video), plays an important role in Internet and multimedia applications, digital appliances such as HDTV, and handheld devices such as digital cameras and mobile phones. Compression allows one to represent images and video with a much smaller amount of data and negligible quality loss. The reduction in data decreases storage requirements (important for embedded devices) and provides higher effective transmission rates (important for Internet enabled devices).

Unfortunately, implementing a compression scheme can be especially difficult. For performance reasons, implementations are typically not portable as they are tuned to specific architectures. And while image and video compression is needed on embedded systems, desktop PCs, and high end servers, writing a separate implementation for every architecture is not cost effective. Furthermore, compression standards are also continuously evolving, and thus compression programs must be easy to modify and update.

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 kernels. As such, image and video compression is a good match for the streaming model of computation, which affords certain advantages in terms of programmability, robustness, and achieving high performance. Our goal is to implement well-known still image and motion picture compression standards—such as JPEG and MPEG-2—in StreamIt [48], a high-level architecture-independent language for the streaming domain. This will result in clean, malleable, and portable codes. In addition, using the stream-aware StreamIt compiler, we can produce highly optimized codes that are competitive with hand-tuned implementations. Our architecture targets include conventional processors, as well as new and emerging wire exposed and multi-core architectures [22, 35, 46, 47, 9].

This work is in the context of the StreamIt programming language [48], an architecture-independent stream language that aims to improve programmer productivity within the streaming domain. StreamIt provides an intuitive programming model, allowing the programmer to build an application by connecting components together into a stream graph, where the nodes represent actors that carry out the computation, and edges represent FIFO communication channels between actors. As a result, the parallelism and communication topology of the application are exposed, empowering the compiler to perform many stream-aware optimizations [1, 20, 30, 42] that elude other languages.

2. MPEG-2 Video Coding and Decoding

MPEG-2 [24] is a popular coding and decoding standard for digital video data. The scheme is a subset of both the DVD-Video [45] standard for storing movies, and the Digital Video Broadcasting specifications for transmitting HDTV and SDTV [17]. The scheme is used by a wide variety of multimedia applications and appliances such as the Tivo Digital Video Recorder [50], and the DirecTV satellite broadcast service [10].

MPEG-2 encoding uses both *lossy* compression 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 color (chrominance information). Humans are also much 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.

Lossless compression eliminates redundant information while allowing for its later reconstruction. Similarities between adjacent video pictures are encoded using motion prediction, and all data is Huffman compressed[23]. The amount of lossy and lossless compression depends on the video data. Common compression ratios range from 10:1 to 100:1. For example, HDTV, with a resolution of 1280x720 pixels and a streaming rate of 59.94 frames per second, has an uncompressed data rate of 1.33 Gigabits per second. It is compressed at an average rate of 66:1, reducing the required streaming rate to 20 Megabits per second [52].

2.1. MPEG Coding

The encoder operates on a sequence of pictures. Each picture is made up of pixels arranged in a 16x16 array known as a macroblock. Macroblocks consist of a 2x2 array of blocks (each of which contains an 8x8 array of pixels). There is a separate series of macroblocks for each color channel, and the macroblocks for a given channel are sometimes downsampled to a 2x1 or 1x1 block matrix. The compression in MPEG is achieved largely via motion estimation, which detects and eliminates similarities between macroblocks across pictures. Specifically, the motion estimator calculates a motion vector that represents the horizontal and vertical displacement of a given macroblock (i.e., the one being encoded) from a 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-code (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 human visual perception.

The DCT 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 then run-length encoded by emitting each non-zero coefficient, followed by the number of zeros that precede it, along with the number of bits needed to represent the coefficient, and its value. The run-length encoder scans the DCT matrix in a zig-zag order (Figure 2) to consolidate the zeros in the matrix.

Finally, the output of the run-length encoder, motion vector data, and other information (e.g., type of picture), are Huffman coded to further reduce the average number of bits per data item. The compressed stream is sent to the output device.

2.2. MPEG Decoding

An MPEG-2 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 are intended to assist scene cuts, random access, fast forward, or fast reverse playback [24, p. 14]. 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} 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 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.

As with the encoding process, pictures are divided up into 16x16 pixel macroblocks, themselves composed of 8x8 blocks. Macroblocks specify colors using a *luminance* channel to represent saturation (color intensity), and two *chrominance* channels to represent hue. MPEG-2 streams specify a chroma format which allows the chrominance data to be sampled at a lower rate. The most common chroma format is 4:2:0 which represents a macroblock using four blocks for the luminance channel and one block for each of the two chrominance channels.

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) maps the frequency matrices to the spatial domain.

Finally, the motion vectors parsed from the data stream are passed to a motion compensator, which reconstructs the original pictures. In the case of I pictures, the compensator need not make any changes

```
int->int filter ZigZagDescramble(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();
    }
  }
}
```

Figure 1. Example filter implementing zig-zag descrambling.

```
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 2. MPEG-2 zig-zag descrambling order.

since these pictures were not subject to motion estimation[1]. In the case of P and B pictures however, motion vectors are used to find the corresponding region in the current reference pictures. The compensator then adds the relevant reference macroblocks to the current picture to reconstruct it. These pictures are then emitted to an output device.

3. StreamIt Programming Language

StreamIt [48] 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). Filters contain a work function that 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 akin to a class in object oriented programming with the work function serving as the main method. 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 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 a 8x8 matrix. An instantiation of a filter can specify the matrix dimensions, as well as the desired ordering. In MPEG, there are two possible scan orders. The `Order` parameter can define the specific scan pattern that is desired. For example, the filter shown in Figure 1 implements the default MPEG-2 scan pattern shown in Figure 2.

[1] I pictures are allowed to contain concealment motion vectors which aid in macroblock reconstruction should a bitstream error destroy the frequency coefficient data. We ignore this special case.

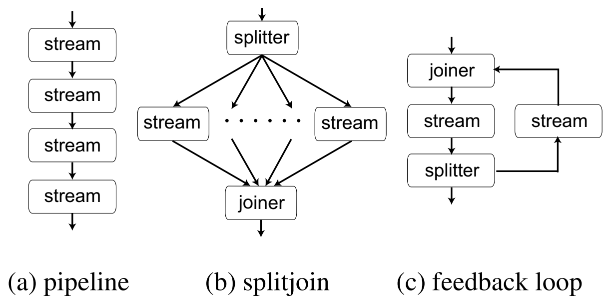


Figure 3. Hierarchical streams in StreamIt.

```
int->int pipeline Decode()
{
  int Order[64] = {...};
  add ZigZagDescramble(64, Order);
  add IQ();
  add IDCT(8, 8);
}
```

Figure 4. Example MPEG decoder pipeline.

In this example, the DCT matrix is represented as a unidimensional stream. The filter peeks or inspects the elements and copies them to the output stream in the specified order. Once all the DCT coefficients are copies, the input stream is deallocated from the tape with a series of pops.

3.2. Hierarchical Streams

In StreamIt, the application developer focuses on the hierarchical assembly of the stream graph and its communication topology, rather than on the explicit management of the data buffers between filters. StreamIt provides three hierarchical structures for composing filters into larger stream graphs (see Figure 3).

Pipeline. The pipeline stream construct composes streams in sequence, with the output of one connected to the input of the next. An example of a pipeline appears in Figure 4. A pipeline is a single input to single output stream. The decoding pipeline in the figure consists of three streams. The first is a filter which zig-zag unorders the input stream, and prepares the data for the inverse quantization and DCT. The output of the filter is consumed by a stream named `IQ` which is a pipeline itself (not shown). This example illustrates the hierarchical nature of stream composition in StreamIt. The `IQ` pipeline performs the inverse quantization, and produces an output stream that is in turn consumed by another stream which performs the inverse DCT. As in the case of a filter, pipelines are also parameterizable.

The `add` keyword in StreamIt constructs the specified stream using the input parameters. 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.

Split-Join. The splitjoin stream 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. For the former, incoming data are replicated to every sibling connected to the splitter. For the latter, data are scattered in a roundrobin manner, with each item sent to exactly one child stream, in order. The splitter type and the weights for distributing data to child streams are declared as part of the syntax (e.g., `split duplicate` or `split roundrobin(`$w_1, \ldots, w_n$`)`). The splitter

```
// N = macroblock size + motion vector data size;
// W = picture width (in pixels);
// H = picture width (in pixels);

int->int splitjoin YCrCbDecoding(int N, int W, int H)
{
  // 4:2:0 chroma format
  split roundrobin(4*N, 1*N, 1*N);

  // last two parameters indicate
  // necessary upsampling in x-y directions
  add LuminanceChannel  (W, H, 0, 0);
  add ChrominanceChannel(W, H, 2, 2);
  add ChrominanceChannel(W, H, 2, 2);

  join roundrobin(1, 1, 1);
}
```

Figure 5. Example MPEG decoder splitjoin.

counterpart is the joiner. It is a specialized filter with multiple input channels but only one output channel. The joiner gathers data from its predecessors in a roundrobin manner (declared as part of the syntax) to produce a single output stream.

The splitjoin stream is a convenient and natural way to represent parallel computation. For example, when the decoder performs the luminance and chrominance channel processing, the computation can occur in parallel. In StreamIt, this is expressed as shown in Figure 5. The input stream contains the macroblock data along with the parsed motion vectors. The data is partitioned and passed to one of three decoding channels, with $4N$ items assigned to the first stream, N items to the second, and N items to the third. The three streams reconstruct the original pictures with respect to the different color channels, and their output is combined by the joiner to produce the final decoded picture.

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 may be used in the MPEG encoder.

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 teleported to the appropriate streams. In StreamIt, such communication is conveniently accomplished using teleport messaging [49].

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 6, the `InverseDCQuantizer` declares a message handler that adjusts its precision (lines 27-29). The `Parser` calls this handler through a *portal* (line

```
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  }
```

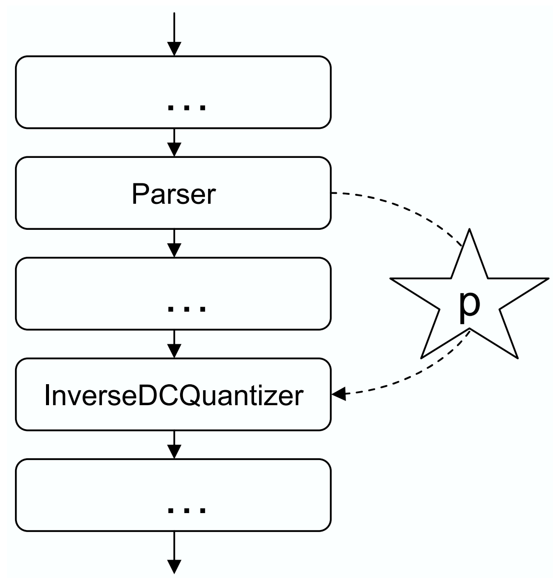


Figure 6. MPEG messaging example.

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.

The interesting aspects of teleport messaging are the semantics for the message latency. Intuitively, the message semantics can be thought of in terms of attaching tags to data items. If the `Parser` sends a message to a downstream filter 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 with the same message. The message flows through the graph until the tagged data reaches its 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. The intuition for upstream messages is similar.

Teleport messaging exposes the true information flow, and avoids the muddling of data streams with control-relevant information. Teleport messaging thus separates the concerns of the programmer from that of a system implementation, thereby allowing the compiler to deliver the message in the most efficient way for a given architecture. Teleport messaging also offers other powerful control over timing and latency beyond what is used in this example [49].

4. MPEG Decoder in StreamIt

We implemented an MPEG-2 decoder in StreamIt. It is a fully portable implementation in that the application is not architecture dependent. The implementation was carried out by one student programmer with no prior understanding of MPEG. The development spanned eight weeks from specification [24] to the first fully functional MPEG decoder. The StreamIt code is nearly 4,921 lines of code with 48 static streams. The MPEG stream parser is the largest single filter, consisting of 1,924 lines of code. The 48 static streams are compiled to 2,150 filters for a picture resolution of 352x240. In contrast, the reference C implementation [53] is nearly 9,832 lines of code, although it provides several features such as interlacing and multi-layer streams that are not yet implemented in the StreamIt decoder.

A noteworthy aspect of the StreamIt implementation is its malleability. We illustrate this using two specific examples. In the first example, we focus on the video sampling rates. MPEG-2 streams are encoded using a 4:2:0 sampling rate, which achieves a 50% reduction in the number of bits required to represent a video, with little noticeable loss of color information. However, better quality is possible with higher sampling rates since more color information is retained from the original picture. In this paper, we describe how our decoder implementation, originally designed to deal with a 4:2:0 sampling rate is modified for a 4:2:2 sampling rate.

In the second example, we describe a straight forward language-level transformation that exposes the data-parallelism across macroblocks in a picture. This is done in the context of the decoder pipeline which consists of the inverse quantization, inverse DCT, and motion compensator. We show that parallelism can be exposed at various levels in the decoding process, from macroblock to block granularities, and that the migration path is trivial.

4.1. Video Sampling Rate

Macroblocks specify colors using a luminance channel to represent saturation (color intensity), and two chrominance channels to represent hue. The human eye is more sensitive to changes in saturation than changes in hue, so the chrominance channels are frequently compressed by downsampling the chrominance data within a macroblock. The type of chrominance downsampling an MPEG-2 encoder uses is its *chrominance format*. The most common chrominance format is 4:2:0, which uses a single block for each of the chrominance channels, downsampling each of the two channels from 16x16 to 8x8. An alternate chrominance format is 4:2:2. It uses two blocks for each chrominance channel, downsampling each of the channels from 16x16 to 8x16. The two chrominance formats are shown in Figure 7.

To support the 4:2:2 chrominance format in our StreamIt decoder, we modified 31 lines and added 20 new lines. Of the 31 modified lines, 23 were trivial modifications to pass a variable representing the chrominance format as a stream parameter. The greatest substantial change was to the decoding splitjoin previously illustrated in Figure 5. In the case of a 4:2:2 sampling rate, the chrominance data, as it appears on the input tape, alternates between each of the two chrominance channels. Thus, a two-tiered splitjoin is used to properly recover the appropriate chrominance channels. The new splitjoin is shown in Figure 7.

4.2. Motion Compensation

An MPEG decoder accepts a bitstream as input and performs Huffman and variable run-length decoding (VLD). This process results in a set of quantized, frequency-domain macroblocks and corresponding motion vectors. The decoder inversely quantizes (IQ) the macroblocks and then performs an inverse DCT (IDCT) to convert the macroblocks to the spatial domain. For predictively coded macroblocks (e.g., P

```
// N = macroblock size + motion vector data size;
// W = picture width (resolution in pixels);
// H = picture width (resolution in pixels);

int->int splitjoin(int chroma) {
  int xsample, ysample; // upsampling requirement

  if (chroma == 420) {  // 4:2:0 chroma format
    split roundrobin(4*N, 2*N);
    xsample = ysample = 2;
  } else {              // 4:2:2 chroma format
    split roundrobin(4*N, 4*N);
    xsample = 2;
    ysample = 0;
  }

  add LuminanceChannel(W, H, 0, 0, chroma);

  add int->int splitjoin {
    split roundrobin(N, N);
    add ChrominanceChannel(W, H, xsample, ysample, chroma);
    add ChrominanceChannel(W, H, xsample, ysample, chroma);
    join roundrobin(1, 1);
  }

  join roundrobin(1, 2);
}
```

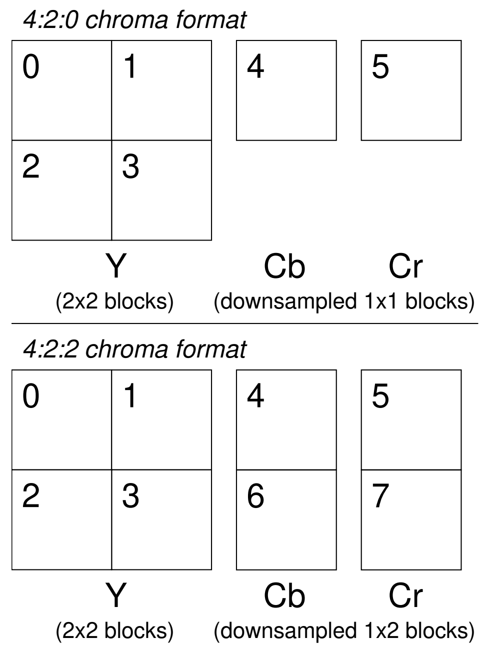


Figure 7. Decoding stream to handle 4:2:0 and 4:2:2 chroma formats. Figures on right illustrate how macroblock orderings differ.

and B pictures), the decoder performs motion compensation (MC) using the input motion vectors to find a corresponding macroblock in a previously decoded, stored reference picture. This 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 future reference. Figure 8 illustrates the decode sequence.

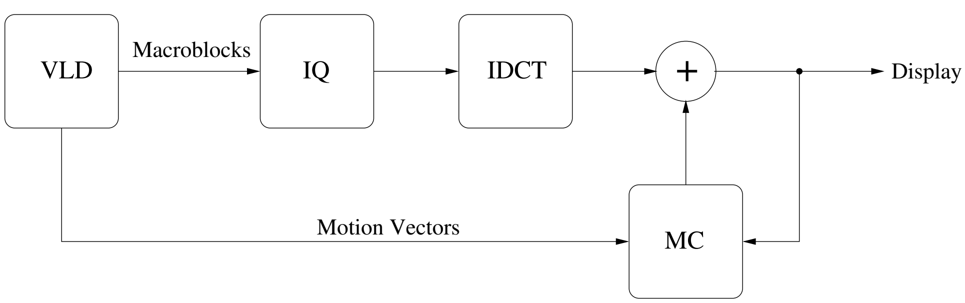
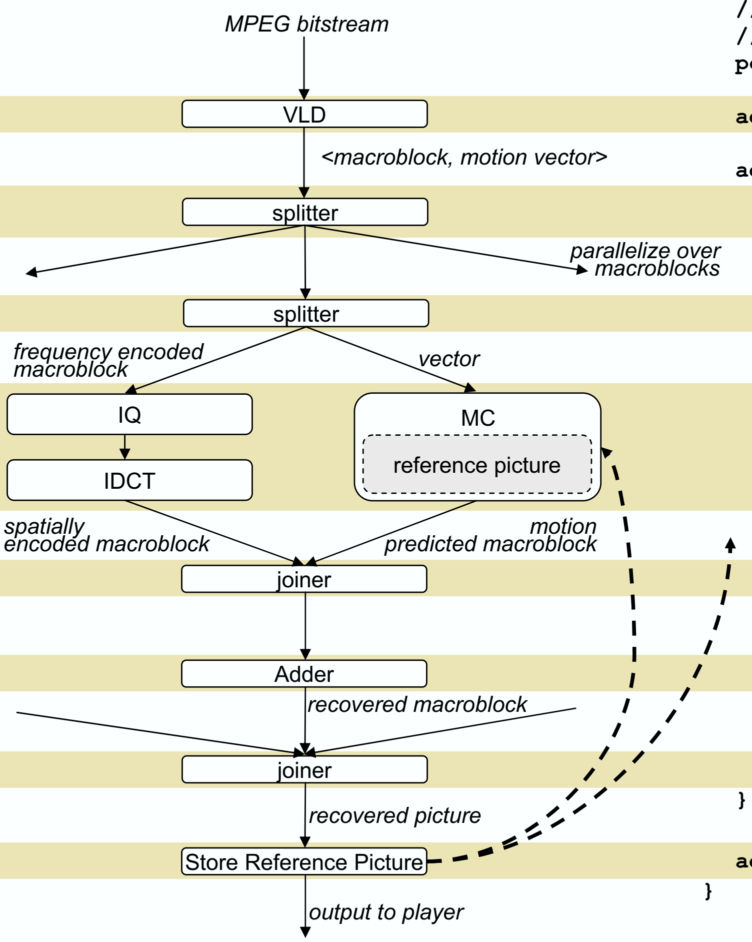


Figure 8. Block diagram of MPEG-2 decode.

A simple strategy for parallelizing the MPEG-2 decoding can exploit the data parallelism among macroblocks. Using this scheme, the Huffman and run-length decoding is inherently serial, as macroblock boundaries can only be discovered by performing the decode operation. Once this decode is complete, a parallel implementation can distribute macroblocks to independent streams (using a splitjoin). Each stream performs the inverse quantization, inverse discrete cosine transform, and motion compensation. Furthermore, each stream locally stores reference macroblocks for future motion compensation. Using this strategy, the streams can execute independently with one exception.




```
bit->int pipeline MPEG Decoder {
  // B = macroblock size;
  // V = motion vector size;
  // W,H = picture width,height
  portal<MC> teleport;

  add VLD();

  add splitjoin {
    split roundrobin(B+V);
    for (int i = 0; i < (W*H)/(16*16); i++) {
      add pipeline {
        add splitjoin {
          split roundrobin(B, V);

          add pipeline {
            add IQ();
            add IDCT();
          }
          add MC() to teleport;

          join roundrobin(B, B);
        }
        add Adder();
      }
    }
    join roundrobin(B);
  }
  add StoreReferencePicture(teleport);
}
```

Figure 9. MPEG-2 decoder exploiting macroblock-level parallelism.

This exception occurs when a stream is performing motion compensation and the corresponding motion vector indicates a reference macroblock stored in some other stream. In this case, inter-stream communication is required to send the reference data to the requesting stream. This situation is not uncommon, and is more prevalent for higher resolution pictures. A simple scheme for handling this situation is for every stream to broadcast its decoded macroblocks to all other streams. This solution has the benefit of being conceptually easy to understand and implement. StreamIt allows programmers to naturally expose such parallelism. A StreamIt pipeline that operates at macroblock granularity is shown in Figure 9. It is worthy to note that there is a high correlation between the stream graph, and the StreamIt syntax describing the pipeline.

The implementation can be made more fine grained by exposing the intra-macroblock parallelism. For example, the IQ-IDCT pipeline can operate at a block level, rather than at a macroblock granularity. This is easily achieved by encapsulating the IQ-DCT pipeline within a splitjoin to scatter the blocks, operate, and gather the results to recover the parent macroblock.

There are many implementation strategies for the decoder, each with varying degrees of exposed parallelism. Of the greatest advantage of the StreamIt implementation is its malleability. The stream graph is easily reconfigured to operate at picture-level granularity (exposing parallelism between chroma channels), macroblock level (exposing even more data-level parallelism), or even at block level (exposing the greatest amount of data-level parallelism). The modularity of the language also affords the ability to cleanly define stream interfaces, and reuse existing components. As an example, the zig-zag descrambler, inverse quantizer, and inverse DCT components were all reused for our JPEG codec implementation. The modularity also reduces the complexity of the debugging process, as stream components can be functionally verified independently, leading to greater programmer productivity.

5. MPEG Encoder in StreamIt

Our MPEG-2 decoder is fully implemented and functional. We are also implementing an MPEG-2 encoder in StreamIt. To date, our encoder is functional although it does not yet implement motion estimation. Thus, the encoder does not achieve any compression of the input video stream. The implementation of the motion estimator is our current focus.

Our implementation of the MPEG-2 encoder exploits data parallelism among macroblocks, as in the case of the decoder. In terms of the motion estimation, each stream searches through a stored reference picture to find the best match for the macroblock it is encoding. The MPEG-2 standard does not specify a minimum or maximum search window for motion estimation: it may be as large as the entire reference picture or as small as a few pixels. In the case where the encoder searches the entire reference picture, every stream requires its own private copy of the reference picture. In StreamIt, teleport messaging serves to broadcast the reference pictures to the encoding streams.

An alternate implementation can exploit the fact that the MPEG-2 standard does not specify a maximum search window, and thus, it can impose a maximum length on the search area, thereby restricting the length of the motion vectors produced by our encoder. The upper bound on motion vector lengths limits the number of reference macroblocks that need to be stored for motion estimation.

In StreamIt, the maximum search window size is modeled with a stream parameter which can be tuned according to various criteria. Larger search windows have the potential for finding better matches and thus a more compact encoding. Smaller search windows limit the amount of communication, offering faster performance at the cost of either less compression or lower quality images. The ideal window size depends on the number of streams, the size of the pictures, and the relative importance of encoder speed and output quality. A programmable StreamIt MPEG-2 encoder thus affords significant flexibility.

6. Related Work

Video codecs such as MPEG-2 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 [33], GRAPE-II [31], and COSSAP [29]. The Synchronous Dataflow model (SDF) represents computation as an independent set of actors that communicate at fixed rates [32]. 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 [38, 39]. 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 [28] and a "dynamic graph topology" to extend compile-time scheduling optimizations to each runtime possibility [27]. Neuendorffer and Lee also extend SDF to support hierarchical parameter reconfiguration, subject to semantic constraints [37]. 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 [51] and process algebras [41].

There are a number of stream-oriented languages besides StreamIt, drawing from functional, dataflow, CSP and synchronous programming styles [44]. Synchronous languages which target embedded applications include Esterel [7], Lustre [21], Signal [19], Lucid [5], and Lucid Synchrone [11]. Additional languages of recent interest are Cg [36], Brook [8], Spidle [12], StreamC/KernelC [26], Occam[13], Parallel Haskell [4] and Sisal [18]. The primary differences between StreamIt and these languages are (i) StreamIt supports (but is no longer limited to) the Synchronous Dataflow [32] 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. [43] 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) [15], 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 [14] and MPEG-2 [16]. Other high-performance programming models for MPEG-2 include manual conversion of C/C++ to SystemC [40], manual conversion to POSIX threads [34], and custom mappings to multiprocessors [2, 25]. 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 chroma sampling rates. In addition, we showed that the StreamIt language is a good match for representing the MPEG stream flow in that there is direct correlation between a 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. In addition, we have prototyped an MPEG-2 encoder, and our current focus is geared toward augmenting the implementation using various motion estimation techniques.

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.

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*, 27:823–846, 2001.

[3] I. Ahmad, Y. He, and M. L. Liou. Video compression with parallel processing. *Parallel Computing*, 28:1039–1078, 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):519–526, 1977.

[6] I. Assayad, P. Gerner, S. Yovine, and V. Bertin. Modelling, Analysis and Parallel Implementation of an On-line 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. *Science of Computer 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] D. Burger, S. Keckler, K. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, and W. Yoder. Scaling to the End of Silicon with EDGE Architectures. *IEEE Computer*, 37(7):44–55, 2004.

[10] L. W. Butterworth. Architecture of the first US direct broadcast satellite system. In *Proceedings of the IEEE National Telesystems Conference*, 1994.

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

[12] 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 Programming and Component Engineering*, 2003.

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

[14] E. de Kock. Multiprocessor Mapping of Process Networks: A JPEG Decoding Case Study. In *Proc. of the 15th Int. Symp. on System Synthesis*, pages 68–73, 2002.

[15] 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 *37th Conference on Design Automation*, 2000.

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

[17] Implementation Guidelines for the use of MPEG-2 Systems, Video and Audio in Satellite, Cable and Terrestrial Broadcasting Applications. ETSI ETR 154, Revision 2, 2000.

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

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

[20] 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.

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

[22] H. P. Hofstee. Power Efficient Processor Architecture and The Cell Processor. In *HPCA*, 2005.

[23] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proc. of the IRE*, 40(9):1098–1101, Sept. 1952.

[24] 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.

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

[26] 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.

[27] D.-I. Ko and S. S. Bhattacharyya. Dynamic Configuration of Dataflow Graph Topology for DSP System Design. In *Proc. of the Int. Conf. on Acoustics, Speech, and Signal Processing*, pages 69–72, 2005.

[28] D.-I. Ko and S. S. Bhattacharyya. Modeling of Block-Based DSP Systems. *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 40(3):289–299, 2005.

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

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

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

[32] E. Lee and D. Messershmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Trans. on Computers*, C-36(1):24–35, January 1987.

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

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

[35] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, and M. Horowitz. Smart Memories: A Modular Reconfigurable Architecture. In *ISCA*, 2000.

[36] 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.

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

[38] 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: Design and Implementation*, 1999.

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

[40] 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 *Proc. of the International Global Signal Processing Conference*, 2004.

[41] 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 *Proc. of the 17th UK Performance Evaluation Workshop*, 2001.

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

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

[44] R. Stephens. A Survey of Stream Processing. *Acta Informatica*, 34(7):491–541, 1997.

[45] J. Taylor. Standards: DVD-video: multimedia for the masses. *IEEE MultiMedia*, 6(3):86–92, July–Sept. 1999.

[46] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffmann, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs. *IEEE Micro*, 2002.

[47] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. In *ISCA*, 2004.

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

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

[50] What Codecs Are Supported to Play TiVoToGo Files on My PC? http://www.tivo.com/codec/.

[51] 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.

[52] B. Vasudev and K. Konstantinos. *Image and Video Compression Standards*. Kluwer, 1997.

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