Stream Programming for Image and Video Compression

by

Matthew Henry Drake

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 26, 2006

© Matthew Henry Drake, MMVI. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.

Author . . . .
Department of Electrical Engineering and Computer Science
May 26, 2006

Certified by . . . .
Saman Amarasinghe
Associate Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Stream Programming for Image and Video Compression

by

Matthew Henry Drake

Submitted to the Department of Electrical Engineering and Computer Science
on May 26, 2006, in partial fulfillment of the
requirements for the Degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Video playback devices rely on compression algorithms to minimize storage, transmission bandwidth, and overall cost. Compression techniques have high realtime and sustained throughput requirements, and the end of CPU clock scaling means that parallel implementations for novel system architectures are needed. Parallel implementations increase the complexity of application design. Current languages force the programmer to trade off productivity for performance; the performance demands dictate that the parallel programmer choose a low-level language in which he can explicitly control the degree of parallelism and tune his code for performance. This methodology is not cost effective because this architecture-specific code is neither malleable nor portable. Reimplementations must be written from scratch for each of the existing parallel and reconfigurable architectures.

This thesis shows that multimedia compression algorithms, composed of many independent processing stages, are a good match for the streaming model of computation. Stream programming models afford certain advantages in terms of programmability, robustness, and achieving high performance.

This thesis intends to influence language design towards the inclusion of features that lend to the efficient implementation and parallel execution of streaming applications like image and video compression algorithms. Towards this I contribute *i*) a clean, malleable, and portable implementation of an MPEG-2 encoder and decoder expressed in a streaming fashion, *ii*) an analysis of how a streaming language improves programmer productivity, *iii*) an analysis of how a streaming language enables scalable parallel execution, *iv*) an enumeration of the language features that are needed to cleanly express compression algorithms, *v*) an enumeration of the language features that support large scale application development and promote software engineering principles such as portability and reusability. This thesis presents a case study of MPEG-2 encoding and decoding to explicate points about language expressiveness. The work is in the context of the StreamIt programming language.

Thesis Supervisor: Saman Amarasinghe
Title: Associate Professor

Acknowledgments

Though we are not now that strength which in old days

Moved earth and heaven; that which we are, we are;

One equal temper of heroic hearts,

Made weak by time and fate, but strong in will

To strive, to seek, to find, and not to yield.

Tennyson, *Ulysses* (1842)

My thanks go first to Rodric Rabbah and William Thies for their direction, advice, and innumerable hours of help; I could not have asked for better role models and guides to the world of computer science research. I thank my advisor Saman Amarasinghe for supporting and guiding this thesis and demanding the highest of standards.

I am grateful to everyone in the StreamIt group, including William Thies, Rodric Rabbah, Michael Gordon, Allyn Dimock, and Janis Sermulins, for their extensive work on the StreamIt language and compiler. Without their help this thesis would not have been possible.

I want to thank my parents for twenty two years spent instilling in me the importance of education and hard work - particularly, my father's early encouragement of my programming interests and my mother's demand for impeccable writing. I owe a huge debt of gratitude to my grandfather, Benjamin Turner, and Meng Mao, for their encouragement and interest in my work.

This thesis extends from a paper[19] by the author, Rodric Rabbah, and Saman Amarasinghe that appeared in the proceedings of the IEEE 2006 International Parallel and Distributed Processing Symposium. The StreamIt project is supported by DARPA grants PCA-F29601-03-2-0065 and HPCA/PERCS-W0133890, and NSF awards CNS-0305453 and EIA-0071841.

Contents

1 Introduction **15**

2 The MPEG-2 Motion Picture Compression Standard **19**

2.1 Compression Techniques 20

2.2 Picture Organization 22

2.3 Temporal Compression 23

2.4 Spatial Compression 24

2.5 Required Block Decoding Components 27

2.6 Video Organization 28

2.7 Additional MPEG-2 Features 28

3 Background and Related Work **31**

3.1 Modeling Environments 33

3.2 Stream Languages 34

3.3 Parallelization Frameworks 34

3.4 Implementation Studies 35

4 The StreamIt Programming Language **37**

4.1 Filters as Programmable Units 38

4.2 Hierarchical Streams 40

4.2.1 Pipeline 40

4.2.2 Splitjoin 41

4.3 Execution Model 42

4.4 Teleport Messaging . . . 43
4.5 Prework Declarations . . . 46
4.6 Dynamic Rates . . . 46
4.7 Functions . . . 48

5 MPEG-2 Codec Implementation 49
5.1 Decoder Structure . . . 49
5.2 Encoder Structure . . . 51
5.3 Motion Estimation . . . 53
5.4 Implementation Statistics . . . 54

6 Programmability and Productivity 57
6.1 Buffer Management . . . 58
6.2 Pipelines Preserve Block Structure . . . 60
6.3 Natural Exposure of Data Distribution . . . 61
6.4 Code Malleability . . . 63
6.5 Hierarchical Construction vs Functional Calls . . . 66
6.6 Usefulness of Teleport Messaging . . . 67

7 Expressing Parallelism 71
7.1 Splitjoins Express Data Parallelism . . . 71
7.2 Hierarchical Streams Expose High Degrees of Parallelism . . . 72
7.3 Parallelizing Motion Prediction . . . 73
7.4 Improving Decoder Parallelization . . . 76
7.5 Performance Results . . . 78

8 StreamIt Limitations and Proposed Extensions 83
8.1 Bitstream Parsing . . . 83
8.2 Functions with Tape Access . . . 86
8.3 Messaging Interfaces . . . 87
8.4 Programmable Splitjoins . . . 88
8.5 Named Work Functions . . . 91

8.6 Stream Graph Reinitialization 93
8.7 Stream Graph Draining 93

9 Conclusions 95

9.1 Future Research 96

List of Figures

2-1 High level view of MPEG-2 decoding and encoding. 19

2-2 MPEG-2 picture subcomposition. 22

2-3 Commonly used chroma formats. 23

2-4 Eliminating temporal redundancy through forward motion estimation. 23

2-5 Eliminating temporal redundancy through backward motion estimation. 24

2-6 Sample input and output for a discrete cosine transform. 25

2-7 Example block quantization. 26

2-8 Zigzag ordering of frequency coefficients from low to high. 27

3-1 Sample block diagram for an MPEG-2 decoder. 32

4-1 Simple division filter. 38

4-2 Zigzag descrambling filter. 39

4-3 Hierarchical streams in StreamIt. 40

4-4 Example pipeline. 41

4-5 Example splitjoin. 42

4-6 Sample steady state execution schedule for a pipeline. 43

4-7 Messaging example. 44

4-8 Pseudocode for picture reordering in the decoder. 46

4-9 Sample filter with prework declaration. 47

4-10 Sample filter with dynamic rate declaration. 47

5-1 MPEG-2 decoder block diagram with associated StreamIt code. . . . 50

5-2 MPEG-2 encoder block diagram with associated StreamIt code. . . . 51

5-3 Motion estimation stream subgraph. 53
5-4 StreamIt line counts for decoder-only, encoder-only, and shared library code. 54
5-5 StreamIt declaration counts for decoder-only, encoder-only, and shared library stream components. 54

6-1 Combining the spatially and temporally decoded data in C. 59
6-2 Combining the spatially and temporally decoded data in StreamIt. . . 59
6-3 Block diagram for spatial decoding (from MPEG-2 specification). . . 60
6-4 StreamIt pipeline for spatial decoding. 61
6-5 2D upsampling decomposed into 1D upsampling 62
6-6 Splitjoins for channel upsampling. 62
6-7 C code excerpt for handling 4:2:0 and 4:2:2 chroma formats. 63
6-8 Original StreamIt code excerpt for handling 4:2:0 chroma format only. 64
6-9 StreamIt code excerpt for handling 4:2:0 and 4:2:2 chroma formats. . 64
6-10 Modified subgraph for handling 4:2:0 and 4:2:2 chroma formats. . . . 65
6-11 Simplified call-trace diagram for C decoder. 66
6-12 Important control parameters sent through the decoder using teleport messaging. 67
6-13 Important control parameters sent through the encoder using teleport messaging. 67
6-14 Communication dependencies between functional units in the C code. 68

7-1 Subgraph for a fine grained 2D inverse DCT. 72
7-2 StreamIt code for the fine grained 2D inverse DCT subgraph. 73
7-3 C code for 2D inverse DCT calculation using two 1D transforms. . . . 74
7-4 Stream topology for motion compensation of a single color channel. . 75
7-5 Exposing parallelism between spatial and temporal decoding in an MPEG-2 decoder. 77
7-6 Three versions of the spatial decoding stream graph and their granularities. 79

7-7 Partitioning the spatial decoding stream graph for 16 tiles of Raw. . . 80
7-8 Layout of the fine grained spatial decoding stream graph on the Raw chip. 80
7-9 Scalability of StreamIt spatial decoding pipeline against single tile C baseline. 81

8-1 Code fragment from parser with (left) and without (right) tape accessing external functions. 86
8-2 Inverse quantization subgraph with a duplicate splitter and roundrobin joiner. 89
8-3 Inverse quantization subgraph with switch splitters and joiners. . . . 89
8-4 Motion estimation stream subgraph with a programmable splitter and joiner. 90
8-5 Subgraph for decoding reference pictures and sending them upstream. 91

Chapter 1

Introduction

Image and video compression algorithms are at the heart of a digital media explosion, playing an important role in Internet and multimedia applications, digital appliances, and handheld devices. Multimedia appliances are ubiquitous in everyday life, encompassed by standards such as DVD [59], HDTV [21], and satellite broadcasting [13]. With even budget cell phones capable of playing and recording videos, these appliances continue to increase in pervasiveness. Virtually all video playback devices rely on compression techniques to minimize storage, transmission bandwidth, and overall cost. Compression techniques often contain complex mathematics and demand realtime performance or high sustained throughput. The end of traditional CPU clock scaling means that Von Neumann architectures can no longer meet these performance requirements and parallel implementations for novel system architectures are needed. Parallel implementations increase the complexity of applications.

The parallel programmer picking a language for a compression algorithm implementation faces a dilemma, forced to trade off productivity for performance. Functional languages provide a high degree of malleability but poor performance. Imperative languages with compiler directed parallelism introduce implementation details and the code loses malleability. High performance imperative languages tie implementations to specific architectures or families of architectures. Assembly code provides the greatest performance but is neither malleable nor portable and mini-

mizes programmer productivity.

Current programming practices and performance demands dictate that the parallel programmer choose a low-level language in which he can explicitly control the degree of parallelism and arduously tune his code for performance. This methodology is not cost effective because architecture-specific code is not portable. Reimplementations must be written from scratch for each of the parallel and reconfigurable architectures that exist, such as clusters, multicores, tiled architectures, ASICs and FPGAs. The process is made more challenging by the continuous evolution of compression standards, which are driven by new innovations in a rapidly growing digital multimedia market.

This thesis shows that stream programming is an ideal model of computation for expressing image and video compression algorithms. The three types of operations in a typical compression algorithm — data representation, lossy compression, and lossless compression — are semi-autonomous, exhibit data and pipeline parallelism, and easily fit into a sequence of distinct processing stages. Operations with these properties fit the streaming model of computation, which treats a computation as a series of data transformations performed by actors arranged in well organized topologies. Stream programming models afford certain advantages in terms of programmability, robustness, and achieving high performance. The goal of my thesis is to influence the design of languages towards the inclusion of features that lend to the efficient implementation and parallel execution of streaming applications, such as image and video compression codecs. Towards this goal I make the following contributions:

1. A clean, malleable, and portable implementation of an MPEG-2 encoder and decoder expressed in a streaming fashion.
2. An analysis of how a streaming language improves programmer productivity.
3. An analysis of how a streaming language enables scalable parallel execution.
4. An enumeration of the language features that are needed to cleanly express compression algorithms.

5. An enumeration of the language features that support large scale application development and promote software engineering principles such as portability and reusability.

This work is in the context of the StreamIt programming language. StreamIt reflects a compiler-aware language design. The language constructs allow for the natural expression of computation and the flow of data through a streaming application. This boosts productivity and allows rapid development because functionality maps directly to code. At the same time, the language exposes both coarse-grained and fine-grained parallelism and makes scheduling and communication requirements explicit. The exposed parallelism and communication requirements allow the compiler to perform domain-specific optimizations and enable parallel execution on a variety of architectures, including scalable multicores.

This thesis presents a case study of MPEG-2 encoding and decoding to explicate points about language expressiveness. MPEG-2 codecs are interesting because they are widely used streaming applications with high realtime frame throughput requirements. The author has also implemented JPEG [30] codecs, and found them to express the same types of computations and demand the same language features as MPEG-2 codecs. In the interests of narrowly tailoring the subject and not forcing the reader to learn two compression schemes, the paper limits its discussion to MPEG-2 coding. I mention JPEG only to intimate that the points I make are broader than the MPEG-2 context and apply to the domain of multimedia compression schemes[1].

This thesis is organized as follows. Chapter 2 provides a detailed description of MPEG-2 video compression. This background is necessary to understand the rest of the thesis. Chapter 3 describes related stream programming research. Chapter 4 includes an in-depth introduction to the StreamIt language. Chapter 5 details the implementation of an MPEG-2 decoder and encoder in a stream based language. Chapter 6 illustrates how a stream programming language improves programmer

[1] The MPEG-4 standard is now replacing the MPEG-2 standard; however, the MPEG-4 standard is extremely complicated and a streaming implementation without new language features discussed in this thesis would be infeasible. In the discussion of future work I mention where the new language features are needed.

productivity for video compression codecs. Chapter 7 describes how a streaming codec implementation can expose parallelism to the compiler and gives performance results showing a parallel scalable implementation on a multicore wire-exposed architecture. Chapter 8 emphasizes parts of MPEG-2 which demand new language features for their clean expression. Chapter 9 concludes and discusses future work extensions.

Chapter 2

The MPEG-2 Motion Picture Compression Standard

MPEG-2 [28] is a popular coding standard for digital video. The scheme is a subset of both the DVD-Video [59] standard for storing movies, and the Digital Video Broadcasting specifications for transmitting HDTV and SDTV [21]. The scheme is used by a wide variety of multimedia applications and appliances such as the Tivo Digital Video Recorder [64], and the DirecTV satellite broadcast service [13].

The amount of compression possible 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 [66].

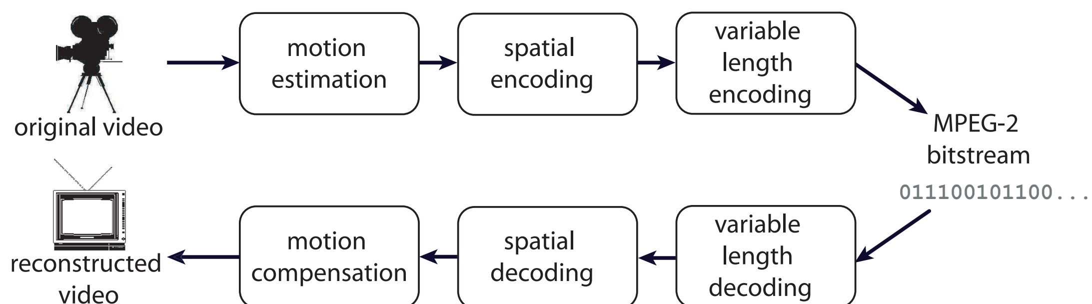


Figure 2-1: High level view of MPEG-2 decoding and encoding.

The MPEG-2 specification contains three types of compression: variable length coding, spatial coding, and motion prediction. Figure 2-1 shows a high level overview of the decoding and encoding process. For a complete description of the MPEG-2 data format and coding scheme one should refer to the official MPEG-2 standard [28]. However, this chapter contains an abbreviated explanation of the standard targeted to a reader lacking prior knowledge of image or video compression algorithms. The explanation focusses on the variable length coding in the parser, the functionality needed for spatial coding and decoding, and the motion prediction components. Iwata et al. estimate that each of these three components constitutes a roughly equal portion of the work needed for decoding [31]. Encoding is similar, although motion estimation constitutes a larger computational effort than the decoder's motion compensation.

This chapter begins with an enumeration of the compression types found in MPEG-2. It then describes picture organization and the temporal and spatial transformations that provide compression. This is followed by a description of video organization and finally a list of optional MPEG-2 format extensions.

2.1 Compression Techniques

MPEG-2 uses both *lossy* and *lossless* compression. Lossless compression eliminates redundant information from a signal while allowing for an exact reconstruction. Lossy compression permanently eliminates information from a picture based on a human perception model. Lossy compression removes details that a casual observer is likely to miss. A lossy compression is irreversible, and a lossy decompression process only approximately reconstructs the original signal. MPEG-2 uses the following compression techniques:

- **Huffman Compression** (*lossless*) Huffman compression [27] is a form of entropy coding. It compresses a signal using variable length codes to efficiently represent commonly occurring subpatterns in the signal.

- **Color Channel Downsampling** (*lossy*) Humans are much better at discerning changes in *luminance*, than changes in *chrominance*. Luminance, or brightness, is a measure of color intensity. Chrominance is a measure of color hue. Pictures are separated into one luminance and two chrominance channels, called the *YCbCr* color space. The chrominance channels are typically downsampled horizontally and vertically.

- **Frequency Quantization** (*lossy*) An image can be expressed as a linear combination of horizontal and vertical frequencies. Humans are much more sensitive to low frequency image components, such as a blue sky, than to high frequency image components, such as a plaid shirt. Unless a high frequency component has a strong presence in an image, it can be discarded. Frequencies which must be coded are stored approximately (by rounding) rather than encoded precisely. This approximation process is called **quantization**. How the different horizontal and vertical frequencies are quantized is determined by empirical data on human perception.

- **Motion Prediction** (*lossless*) Frames of a video contain a great deal of temporal redundancy because much of a scene is duplicated between sequential frames. Motion estimation is used to produce motion predictions with respect to one or more reference frames. Predictions indicate what any given frame should look like. For similar frames, only the motion estimate and any error between the predicted values and the actual values must be encoded.

- **Difference Coding** (*lossless*) Over any given region in an image the average color value is likely to be similar or identical to the average color in surrounding regions. Thus the average colors of regions are coded differentially with respect to their neighbors. Motion information at neighboring regions is also likely to be similar or identical and is therefore coded differentially with respect to motion at neighboring regions.

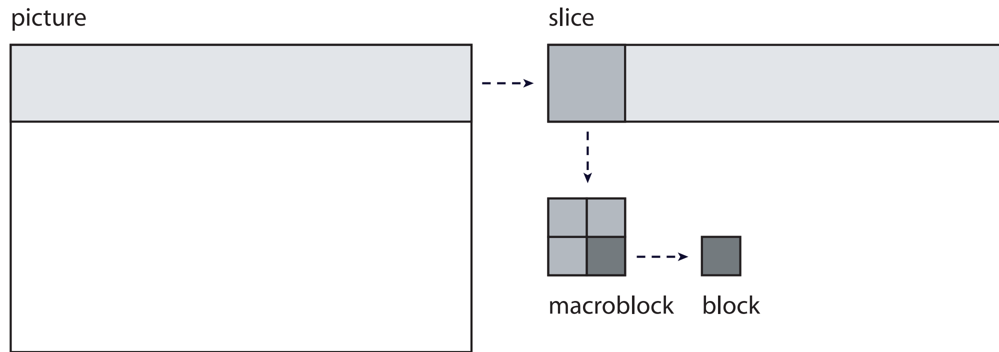


Figure 2-2: MPEG-2 picture subcomposition.

2.2 Picture Organization

Figure 2-2 shows how the MPEG-2 standard organizes pictures. Each picture breaks into 16x16 groups of pixels called **macroblocks**. Adjacent sequences of macroblocks are contained in a structure called a **slice**. Pictures and macroblocks are defined in the YCbCr color space, and the first step of encoding is converting the picture into this color representation.

A macroblock is itself composed of 8x8 subpixel **blocks**. There are always exactly 4 luminance blocks that form a 2x2 array to cover the macroblock. Because of human insensitivity to chrominance information, each of the two chrominance channels may be downsampled.

The type of downsampling in an MPEG-2 stream is called its **chroma format**. The two most common chroma formats are shown in Figure 2-3. The more common of the two is the 4:2:0 format. This format specifies that each chrominance channel in a macroblock be represented by a single block, horizontally and vertically downsampling a macroblock from 16x16 to 8x8 subpixels. A 4:2:0 macroblock therefore contains a total of 6 blocks. An alternate format is 4:2:2. The 4:2:2 format uses two blocks for each chrominance channel, horizontally downsampling each macroblock from 16x16 to 8x16 subpixels. A 4:2:2 macroblock therefore contains a total of 8 blocks. A 4:4:4 chroma format also exists but is not commonly used, and specifies no color channel downsampling, and uses 4 blocks to represent each color channel in a macroblock.

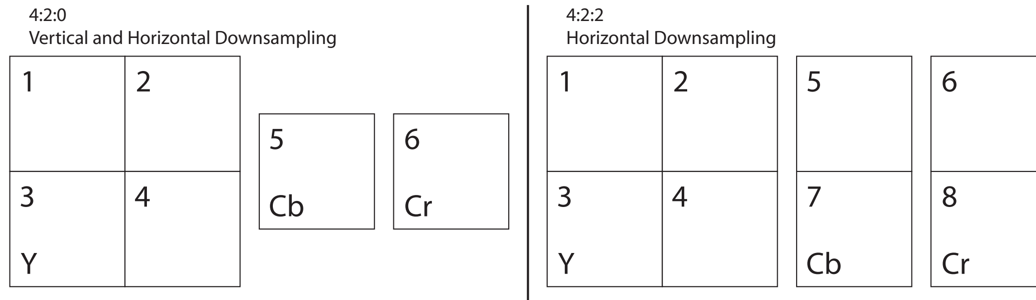


Figure 2-3: Commonly used chroma formats.

2.3 Temporal Compression

Temporal compression in MPEG-2 is achieved via motion prediction, which detects and eliminates similarities between macroblocks across pictures. For any given macroblock M, a motion estimator forms a prediction: a **motion vector** that contains the horizontal and vertical displacement of that macroblock from the most similar macroblock-sized area in one or more reference pictures. The matching macroblock is removed (subtracted) from M on a pixel by pixel basis. The result is a residual **predictive coded (P)** macroblock. The residual macroblock contains the difference between the motion predicted values for the macroblock and the macroblock's actual values. A P macroblock always uses forward motion prediction, meaning that the reference frame precedes it temporally. (See Section 2.6 for more details on picture referencing and organization.) Figure 2-4 illustrates forward motion estimation.

Figure 2-4: Eliminating temporal redundancy through forward motion estimation.

Macroblocks encoded without the use of motion prediction are **intra coded (I)** macroblocks. In addition to the forward motion prediction used by P macroblocks, it is possible to encode new macroblocks using motion estimation from both temporally previous and subsequent pictures. Such macroblocks are **bidirectionally predictive coded (B)** macroblocks, and they exploit a greater amount of temporal locality. A B macroblock may contain two motion vectors, referencing both previous and subsequent pictures; in this case, the motion prediction is an unweighted average of the forward and backward predictions. Figure 2-5 illustrates backward motion estimation.

Figure 2-5: Eliminating temporal redundancy through backward motion estimation.

All blocks in macroblocks, whether intra coded or residually encoded, undergo spatial compression. Except for the first macroblock in a slice, motion vectors are compressed by coding them differentially with respect to the motion vectors in the previously decoded macroblock[1].

2.4 Spatial Compression

Each block undergoes a two-dimensional **Discrete Cosine Transform** (DCT), which is a frequency transform that separates the block into components with varying visual importance. As shown in Figure 2-6, the DCT takes one 8x8 block as input

[1]A second exception is for the first set of motion vectors following an intra coded macroblock. These vectors must always be fully coded because intra coded macroblocks have no motion vectors.

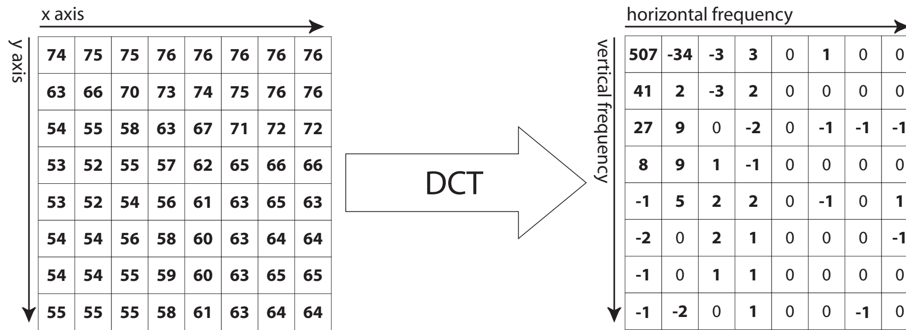


Figure 2-6: Sample input and output for a discrete cosine transform.

and produces a transformed 8x8 block of frequency coefficients as output. Horizontal frequency increases towards the right of the block and vertical frequency increases towards the bottom of the block. The upper left corner of the block contains the lowest frequencies and the lower right corner contains the highest frequencies.

The DCT by itself is lossless[2] but enables the quantization of blocks according to a **quantization table** of **quantization values**, also in the frequency domain. The quantization table reflects a human's relative abilities to discern different frequency components of an image. The quantization table itself may contain any values and can be specified in the MPEG-2 bitstream, although usually one of several standard tables is used. Each value in a frequency-transformed block is divided by the corresponding quantization value, with any remainder thrown away. An example block quantization appears in Figure 2-7[3]. A small error may be introduced to individual frequency components and most low energy frequency components are simply reduced to 0. This stage introduces much of the lossy compression in MPEG-2 coding.

MPEG-2 uses two quantization tables. One table is used for all intra coded blocks and the other for residually coded blocks. At irregular intervals, an MPEG-2 bitstream indicates a **quantization scale code** which provides an additional scaling

[2] I ignore a possible loss of precision, an issue addressed by the MPEG-2 specification and explained subsequently in Section 2.5

[3] The quantization process is technically more complicated than the math I have just described, although the description is conceptually accurate. The output block in the figure is accurately quantized, but cannot be arrived at by the division process I just described.

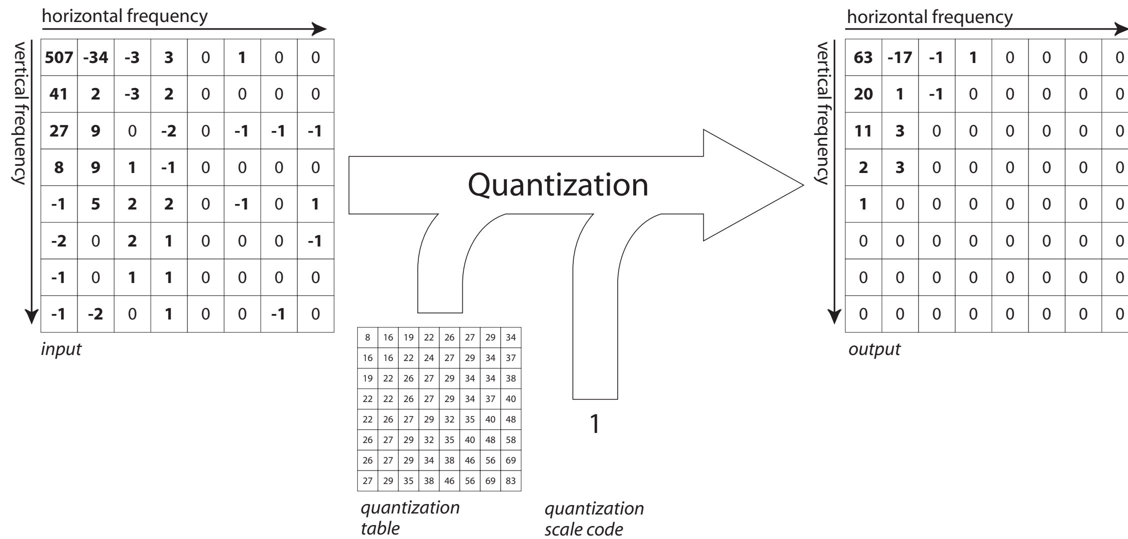


Figure 2-7: Example block quantization.

factor that affects all frequencies in a block. One can adjust the desired compression level and control the video bitrate (bits per second) by tuning the quantization scale code between macroblocks. In an encoder this control is typically realized using feedback about the final entropy coded output bitrate earlier in the quantization stage.

The upper left value in the frequency transformed block contains the **DC coefficient**, which is the coefficient corresponding to the zero frequency in both the horizontal and vertical dimensions. Less formally, this is the average color of the block. MPEG-2 differentially encodes the DC block value for intra coded blocks. The first DC coefficient in the first block in a slice is fully encoded and all subsequent DC coefficients in a slice are differentially coded. Note that the differential coding semantics for DC coefficients and motion vectors guarantee that macroblocks in different slices are coded independently from each other.

After quantization a block is **zigzag** ordered. Zigzag ordering sorts a block's values from lowest to highest frequency. Since low-frequency components are more likely to have non-zero values following quantization, zigzag ordering consolidates non-zero block coefficients together at the beginning of the block. The zigzag order commonly used by MPEG-2 is shown in Figure 2-8

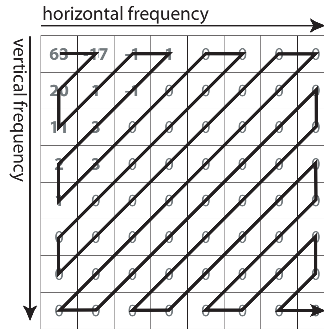


Figure 2-8: Zigzag ordering of frequency coefficients from low to high.

The zigzag ordered data is then Huffman compressed using a set of predefined Huffman tables defined in the MPEG-2 specification. Picture metadata, such as the picture type, changes to the quantization scale code, and motion vectors, are also Huffman encoded and interleaved in the bitstream.

2.5 Required Block Decoding Components

Data transformation pairs such as a DCT and an inverse DCT (IDCT), may accidentally introduce loss of data precision, due to hardware architecture and algorithm differences in a decoder and encoder. While such imprecisions are tiny, the use of temporal compression means that small imprecisions accumulate and magnify over the course of several motion predicted pictures and quickly become noticeable. For this reason the MPEG-2 specification places specific functional constraints on mathematical operations in MPEG-2 codecs:

- The frequency coefficients emitted from the inverse quantization stage must be saturated within predefined levels.
- The low-order bit of the highest frequency value in a block is used as a parity bit on the value of the block. In the encoder this bit is set between the DCT and quantization. In the decoder the bit is checked between the saturated inverse quantization and the IDCT. This setting and checking of the bit is called **mismatch control**.

- The output of the IDCT is saturated within predefined levels.

2.6 Video Organization

Just as macroblocks have an associated I, P, or B type, pictures also have an associated type, used to limit the kinds of macroblocks that they may contain. I pictures contain only I macroblocks, P pictures contain I or P macroblocks, and B pictures may contain I, P, or B macroblocks. Only I and P pictures are used as references for motion prediction and all I and P pictures are automatically considered references. B pictures are never used as references.

The highest level of organization in an MPEG-2 data stream is the **Group of Pictures** (GOP), which contains all the information needed to reconstruct a temporally continuous sequence of video. GOPs consist of I, P, and B pictures. 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, where the subscripts denote the temporal ordering of the pictures in the video:

$$\mathrm{I}_1\ \mathrm{B}_2\ \mathrm{B}_3\ \mathrm{P}_4\ \mathrm{B}_5\ \mathrm{B}_6\ \mathrm{P}_7\ \mathrm{B}_8\ \mathrm{B}_9\ \mathrm{P}_{10}\ \mathrm{B}_{11}\ \mathrm{B}_{12}\ \mathrm{I}_{13}\cdots$$

Any backwards motion vector in a picture refers to the immediately preceding reference picture. Likewise, any forward motion vector refers to the subsequent reference picture. To simplify the decoding process, pictures are not ordered temporally in the data stream, but rather in the order that they are needed for decoding: P pictures are always coded with respect to the previous reference picture and B pictures are always coded with respect to the previous two reference pictures. Thus, the picture pattern previously described is ordered in the MPEG-2 data stream as:

$$\mathrm{I}_1\ \mathrm{P}_4\ \mathrm{B}_2\ \mathrm{B}_3\ \mathrm{P}_7\ \mathrm{B}_5\ \mathrm{B}_6\ \mathrm{P}_{10}\ \mathrm{B}_8\ \mathrm{B}_9\ \mathrm{I}_{13}\ \mathrm{B}_{11}\ \mathrm{B}_{12}\cdots$$

2.7 Additional MPEG-2 Features

Because MPEG-2 targets a wide range of devices, the specification is complicated by additional features that make decoding any given video possible on a range of archi-

tectures. The following features are mentioned for the sake of completeness, but are excluded from the StreamIt MPEG-2 codec implementations. These features constitute alternative data formats, rather than compression or algorithmic enhancements, and are suitable for exclusion in research-oriented MPEG-2 implementations.

- **Interlacing** is a legacy television format needed to support many analog output devices. An interlaced frame contains only half of the original picture data, leaving out alternating horizontal lines. Sequential frames alternate between encoding even and odd scan lines. The alternative to interlacing, which fully encodes each picture, is called **progressive scan**.

- The MPEG-2 bitstream can contain **layers** which contain alternate encodings of the same picture. A motivating example for this feature is the DVD format, which typically encodes an interlaced version of a movie in the primary layer, and an interlaced version containing the alternate scan lines in a secondary layer. Devices that output interlaced pictures can ignore the secondary layer and devices that output progressive pictures can combine the two layers to produce the complete video.

- **Concealment motion vectors** indicate motion estimates for intra-coded macroblocks. These concealment motion vectors are only used to form a macroblock prediction if bitstream errors prevent correct recovery of blocks contained in the macroblock. This plays an important role in the decoding of broadcast MPEG-2 streams such as satellite or HDTV, where transport errors are likely to occur.

Chapter 3

Background and Related Work

Image and video compression codecs belong to the family of streaming applications. These are applications that consume an input stream, perform a set of transformations on the data, and produce an output stream. Streaming applications operate on potentially infinite streams of data, exhibit strong producer-consumer locality, and often have sustained throughput or realtime output requirements.

Previous research includes work in modeling environments, stream languages, parallel computing frameworks, and implementation studies. Modeling environments focus on the expression of stream programs as block topologies and are oriented towards rapid prototyping and programmer efficiency. Stream languages have tried to expose parallelism and communication requirements to the compiler and improve programmer productivity by hiding implementation details. Parallel computing frameworks and implementation studies have tried to produce high performance parallel implementations of video codecs for specific architectures.

My work is in the context of the StreamIt programming language, which tries to provide the best of all worlds: it allows the easy expression of streaming applications, naturally exposes fine-grained and coarse-grained parallelism, and enables a compiler to produce scalable parallel implementations. The summary of related work in modeling environments, languages, and parallel frameworks mentions the differences that set StreamIt apart. StreamIt itself is discussed in the following chapter, with a description of the key features that embody the language's design goals.

Relevant to all stream programming efforts is the notion of a **structured block diagram**. A block diagram illustrates the computations and flow of data through an application. The block diagram's purpose is to provide a clean, conceptual understanding of the application's behavior, cleanly abstracting functionality and ignoring implementation, architecture, and performance oriented details. Boxes represent transformations on the data in the data stream and arrows indicate the flow of data between functional blocks. A sample block diagram for MPEG-2 decoding is shown in Figure 3-1[1].

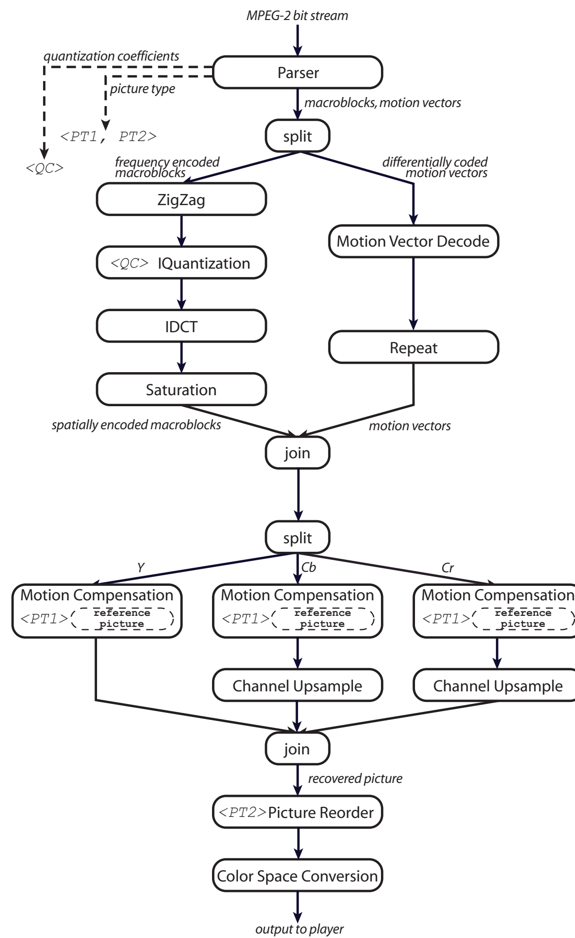


Figure 3-1: Sample block diagram for an MPEG-2 decoder.

[1]There is no official block diagram for MPEG-2 decoding included with the specification, but this is a block diagram I produced based on the specification.

3.1 Modeling Environments

There have been many efforts to develop expressive and efficient models of computation. Ptolemy [42], GRAPE-II [39], COSSAP [37], and MATLAB all provide computing models targeted for rapid prototyping environments. Synchronous Dataflow [41] (SDF) provides a model that affords benefits to efficient scheduling and exposes parallelism and communication. SDF represents computation as a set of independent actors that communicate at fixed rates [41]. Real applications, such as MPEG-2 codecs, have data transforms that dictate sets of behavior changing parameters to other transforms. These parameters occur at irregular intervals and interrupt the regular flow of data. For example, an MPEG-2 bitstream updates the quantization scale code at irregular intervals. Expressing real applications in an SDF model requires extensions that provide dynamic communication and out-of-band control messages.

Synchronous Piggybacked Dataflow (SPDF) supports control messages in the form of a global state table with well-timed reads and writes [47, 48]. SPDF is evaluated using MP3 decoding, and would also be effective for MPEG-2 decoding.

Ko and Bhattacharyya also extend SDF with the dynamism needed for MPEG-2 coding; they use "blocked dataflow" to reconfigure sub-graphs based on parameters embedded in the data stream [36] and a "dynamic graph topology" to extend compile-time scheduling optimizations to each runtime possibility [35].

Neuendorffer and Lee extend SDF to support hierarchical parameter reconfiguration, subject to semantic constraints [46]. These models allow reconfiguration of an actor's I/O rates and require alternate or parameterized schedules.

MPEG-2 encoding has also been expressed in formalisms such as Petri nets [65] and process algebras [50].

3.2 Stream Languages

There are a number of stream-oriented languages drawing from functional, dataflow, CSP and synchronous programming styles [55]. StreamIt is one instance of a stream-based language. Synchronous languages which target embedded applications include Esterel [10], Lustre [26], Signal [23], Lucid [7], and Lucid Synchrone [14]. Other languages of recent interest are Cg [44], Brook [12], Spidle [15], StreamC/KernelC [33], Occam[16], Parallel Haskell [4] and Sisal [22]. StreamIt distinguishes itself from these languages because (*i*) StreamIt supports (but is no longer limited to) the Synchronous Dataflow [41] model of computation, (*ii*) StreamIt offers a "peek" construct that allows an actor to inspect an item on its input channel without consuming it, (*iii*) StreamIt imposes a single-input, single-output hierarchical structure on the stream graph, and (*iv*) StreamIt provides the teleport messaging feature for out-of-band communication, discussed in depth later in Section 4.4.

3.3 Parallelization Frameworks

Video codecs have been a longtime focus of the embedded and high-performance computing communities. Many researchers have developed both hardware and software schemes for parallel video compression; Ahmad et al. [3] and Shen et al. [53] provide reviews. I focus on programming models used to implement image and video codecs 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 [8, 9]. 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 while making the programming concepts as simple as possible.

Another programming model is the Y-Chart Applications Programmers Interface

(YAPI) [18], 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 [17] and MPEG-2 [20].

Other high-performance programming models for MPEG-2 include manual conversion of C/C++ to SystemC [49], manual conversion to POSIX threads [43], and custom mappings to multiprocessors [2, 31]. StreamIt's focus lies on the programmability, providing an architecture-independent representation that is natural for the programmer while exposing pipeline and data parallelism to the compiler.

3.4 Implementation Studies

A number of implementation studies have generated parallel MPEG-2 decoders and encoders. Ahmad et al. have had success with MPEG-2 decoder implementations for distributed memory architectures that expose parallelism at the picture level [5, 6]. They have also developed a multiprocessor MPEG-2 encoder which parallelizes encoding across GOPs [2]. Because GOPs are spread out over a video, this approach has high latency but is suitable for offline encoders or decoders.

Li et al. have produced ALPBench [43], a set of benchmarks that includes both an MPEG-2 encoder and decoder. Both the encoder and decoder have been written to expose parallelism at the slice level. Schneider uses a VHDL-based methodology for modeling multimedia applications [51] and attempts to perform spatial decoding in parallel with temporal decoding. Jacobs et al. have implemented a pipelined MPEG-2 encoder which uses thread level parallelism and distributes sequential stages of the MPEG-2 encoding process to different processors on a shared network [32].

These implementations are useful for describing the types of parallelization that work well for certain architectures. They would be useful for benchmarking a stream language compiler targeted to one of their architectures.

Chapter 4

The StreamIt Programming Language

StreamIt [62] is an architecture independent language that is designed for stream programming. StreamIt tries to preserve the block diagram structure in the program definition, allowing the application developer to express functionality with a one-to-one mapping to code.

StreamIt also makes the parallelism inherent in a computation explicit in the program representation, exposing relevant scheduling and communication details to the compiler. This follows from the preservation of the block diagram structure: each block represents an independent computation kernel which can be executed in parallel with all other blocks. The arrows expose all the communication requirements between the functional blocks and represent FIFO-ordered communication of data over tapes. The programmer can focus on a clean and malleable implementation without worrying about the specific mapping of tasks to hardware. The implementation is portable across a wide range of architectures because the compiler is responsible for determining layout and task scheduling.

StreamIt leverages the SDF model of computation, but also supports dynamic communication rates and out-of-band control messages. Previous StreamIt publications describe the language [62], compiling StreamIt programs for parallel architectures [25, 63], and how StreamIt enables domain-specific [1, 38], cache-aware [52],

and scheduling [34] optimizations.

4.1 Filters as Programmable Units

The fundamental programmable unit in StreamIt is the **filter**. A filter represents an independent actor with a single input and output data channel. Each filter executes atomically in its own private address space. All communication with other filters is via the input and output channels and occasionally via control messages (see Section 4.4). 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. Computations over a sliding window and permutations on input streams are simplified by the peek construct. The `push`, `pop`, and `peek` rates are declared (when known) as part of the work function, allowing the compiler to apply various optimizations and construct efficient execution schedules.

```
int->int filter DivideBy(int divisor) {
    work pop 1 push 1 {
        push(pop()/divisor);
    }
}
```

Figure 4-1: Simple division filter.

The simplest filter definition used in the MPEG-2 decoder is given as an example in Figure 4-1. This filter consumes a stream whose elements are of type `int` and produces a stream of the same type. This filter's output is simply its input divided by the divisor parameter given at instantiation time. (The current version of the StreamIt compiler resolves all such parameters at compile time, allowing additional optimizations.)

The parameterization of filters allows for multiple instantiation with different configurations, facilitating malleability and code reuse. Shown in Figure 4-2 is a

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

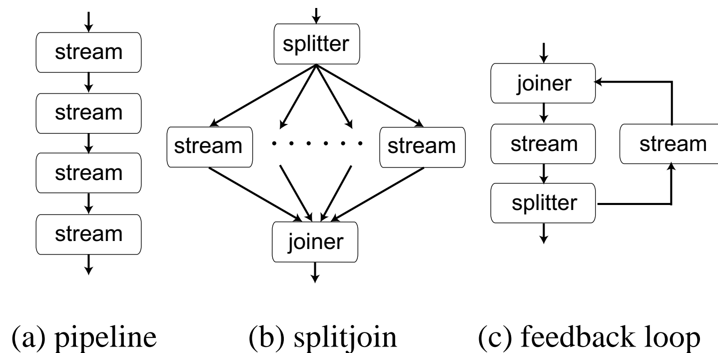
Figure 4-2: Zigzag descrambling filter.

filter that performs an arbitrary reordering on a set of N elements according to an ordering matrix. Each instantiation of the filter specifies the matrix dimensions, as well as the desired ordering. The `Order` parameter defines the specific scan pattern that a filter instance will use. In the example the filter performs the zigzag descrambling necessary to reorder the input stream in the decoder (see Section 2.4). The zigzag scrambling in the encoder reuses this filter with a different `Order` matrix.

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. The input and output buffers are represented implicitly. It has been shown that this program representation enables the automatic generation of vector permutation instructions [45].

4.2 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, shown in Figure 4-3, for composing filters into larger stream graphs. A **pipeline** places components in series, connecting the output from one filter to the input of the subsequent filter. A **splitjoin** places filters in parallel and specifies both a distribution of data and a gathering of data. There is also a **feedback loop** hierarchy which is not needed for MPEG-2 decoding or encoding. All hierarchical stream components are parameterizable. Because each hierarchical component itself consists of a single input and output, hierarchical components may be treated like filters and used inside of increasingly larger stream hierarchies.



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

Figure 4-3: Hierarchical streams in StreamIt.

4.2.1 Pipeline

A pipeline is a single input to single output parameterized stream. It composes streams in sequence, with the output of each filter connected to the input of the subsequent filter. An example pipeline from the MPEG-2 decoder appears in Figure 4-4. This pipeline decodes blocks, compressed as described in Section 2.4. The first filter zigzag reorders the input stream, preparing the data for the inverse quantization and IDCT. The output of the filter is consumed by a stream named `IQuantization` that performs the inverse quantization, and produces an output stream that is in

```
int->int pipeline BlockDecode() {
  int[64] Order = {...};
  add ZigZagUnordering(64, Order);
  add InverseQuantization();
  add Saturation(-2048, 2047);
  add MismatchControl();
  add 2D_iDCT(8); // 8x8 2D IDCT
  add Saturation(-256, 255);
}
```

Figure 4-4: Example pipeline.

turn consumed by another stream that performs `Saturation`, which produces output consumed by the `MismatchControl` filter, which in turn passes the data to the `2D_iDCT` and one final `Saturation`.

The `add` keyword in StreamIt instantiates a specified stream using any given 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 modular design and development of large applications, thereby promoting collaboration, increasing code reuse, and simplifying debugging.

4.2.2 Splitjoin

A splitjoin processes data in parallel, specifying both a data scattering and gathering. In a splitjoin, the **splitter** distributes the data and the **joiner** gathers the data. 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 is a specialized filter with multiple input channels but only one output channel. The joiner gathers data from its predecessors

```
int->int splitjoin SpatialDecoding {
    split roundrobin(64, 16);
    add BlockDecode();
    add MotionVectorDecode();
    join roundrobin(64, 8);
}
```

Figure 4-5: Example splitjoin.

in a roundrobin manner to produce a single output stream.

An example splitjoin is shown in Figure 4-5, which encapsulates the spatial decoding necessary in the MPEG-2 decoder. Its input consists of a stream of data interleaving coded block coefficients and coded motion vectors. The roundrobin splitter separates the data, passing block coefficients to the `BlockDecode` component and motion vectors to the `MotionVectorDecode` component. Note that this splitjoin treats the `BlockDecode` pipeline, previously defined in Figure 4-4, as a primitive element. The roundrobin joiner remerges the data streams. In this case, the `MotionVectorDecode` filter consumes two elements for every element it produces, so the joiner has a different join rate than the splitter. Splitters and joiners may be expressed at the natural granularity of the data and need not have matched rates.

4.3 Execution Model

A StreamIt program can be abstracted as a directed graph in which a node is either a filter, splitter, or joiner, and an edge represents a data channel. When a node executes, it removes data stored on incoming data channels and generates output on outgoing data channels. A single node execution atomically transfers the smallest amount of data across the node.

An execution schedule is an ordered list of executions of nodes in the graph. StreamIt programs have two execution schedules: one for initialization and one for steady state behavior. The initialization schedule primes the input channels, allowing filters with peeking to execute the first instance of their work functions.

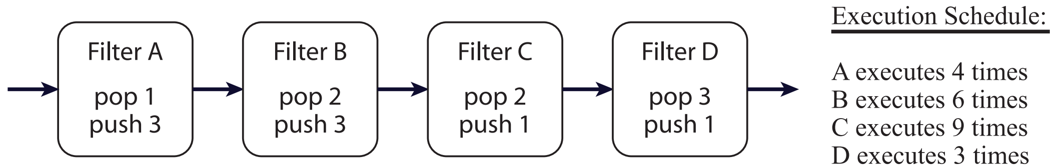


Figure 4-6: Sample steady state execution schedule for a pipeline.

The steady state schedule leaves exactly the same number of data items on all channels between executions of the schedule. Figure 4-6 shows an example of a steady state execution schedule for a pipeline.

The StreamIt compiler can derive steady-state schedules at compile time for portions of a stream graph with statically determined data rates. Nodes with dynamic rates require additional runtime semantics but the conceptual model is still expressed in terms of static execution schedules. A more detailed explanation of the execution model and program scheduling in StreamIt can be found in [34, 56].

4.4 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. It can not easily be intermingled with the block coefficients that the parser outputs because control parameters occur at irregular intervals. In StreamIt, such communication issues are resolved conveniently

```
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 4-7: Messaging example.

using **teleport messaging** [63].

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 4-7, 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 after[1] 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 need not be implemented this way.

The intuition for upstream messages is somewhat similar. Namely, imagine a feedback loop connecting the downstream sender with the upstream message receiver. The downstream filter uses the loop to send tokens on every iteration, and the upstream filter checks the values from the loop before each of its executions. If the value is non-zero, it is treated as a message, otherwise the token is ignored. In this scenario, the upstream message is processed immediately before it generates data that the sender will consume in k of its own iterations.

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 dependencies 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 is similar to the SDF extensions described in Section 3.1. However, control messages in StreamIt are more expressive because they allow up-

[1]This appeared as "immediately before" in the original version of the PPoPP 2005 paper, but has since been updated.

stream messages, adjustable latencies, and more fine-grained delivery (i.e., allowing multiple execution phases per actor and multiple messages per phase).

4.5 Prework Declarations

Many filters require initial data processing before entering a steady-state execution pattern. For instance, an exponentially weighted average might generate its initial value using many data items on its first execution, but only consume a single item per subsequent execution. StreamIt supports this by allowing a `prework` keyword to specify a work function that gets executed once before the regular work function.

The MPEG-2 decoder requires such a filter immediately before outputting a decoded picture sequence. As explained in Section 2.6, the decoder must reorder pictures from their coded order to their temporal order. The pseudocode shown in Figure 4-8 explains conceptually how to reorder the pictures to their temporal order. This pseudocode is easily implemented using a filter with a prework declaration as written in Figure 4-9.

```
store the first (I) picture in a delayed picture buffer;
foreach picture do
    if the picture is a B-picture then
        output immediately
    else
        output the picture in the delayed picture buffer and store the current
        picture into the buffer
    end
end
```

Figure 4-8: Pseudocode for picture reordering in the decoder.

4.6 Dynamic Rates

Not all filters are amenable to static (initialization-time) work rate declarations. For example, filters which implement entropy coding/decoding schemes, such as Huffman coding, will have an input / output ratio that is data dependent. Support

```
int->int filter PictureReorder(int picture_size) {
    int[picture_size] databuffer;
    int next_picture_type;

    prework pop datarate {
        for (int i = 0; i < datarate; i++)
            databuffer[i] = pop();
    }

    work pop datarate push datarate {
        if (next_picture_type == B_PICTURE) {
            // B-picture is next
            for (int i = 0; i < datarate; i++)
                push(pop());
        } else {
            // I or P picture is next
            for (int i = 0; i < datarate; i++) {
                push(databuffer[i]);
                databuffer[i] = pop();
            }
        }
    }

    handler setPictureType(int _next_picture_type) {
        next_picture_type = _next_picture_type;
    }
}
```

Figure 4-9: Sample filter with prework declaration.

```
int->int filter RunLengthDecoder {
    work pop 2 push * {
        int itemQuantity = pop();
        int itemValue = pop();
        for (int i = 0; i < itemQuantity; i++)
            push(itemValue);
    }
}
```

Figure 4-10: Sample filter with dynamic rate declaration.

for these dynamic rate filters is provided by letting a wildcard `*` symbol indicate an unknown rate. A simple example is a run length decoder, shown in Figure 4-10. Note that for most dynamic rate filters, there are usually at least two equally reasonable formulations, either a static input and dynamic output rate or a dynamic input and static output rate.

4.7 Functions

StreamIt currently supports external functions and **helper** functions which facilitate modularity and code reuse. External functions typically come from a math library and examples used in the compression implementations are `round`, and `floor`. Helper functions are limited in scope to the filter in which they are defined, and are important for the same reasons as in procedural languages. They are particularly useful as a means to breakup code within filters which represent particularly complex computations that are not easily subdivided. Large filters responsible for complicated actions such as motion prediction or estimation rely heavily on helper functions to improve code readability by abstracting functionality.

Chapter 5

MPEG-2 Codec Implementation

This chapter describes the MPEG-2 decoder and encoder implementations written in StreamIt. Sections 5.1 through 5.3 explain the actual code base and program structure. Section 5.4 describes the specific functionality in the StreamIt implementations and highlights implementation statistics pertinent to the discussion of programmability and parallelism.

5.1 Decoder Structure

Figure 5-1 shows the MPEG-2 decoder pipeline, correlated with the StreamIt code. A few simplifications have been made to the code and the figure for purposes of explanation[1]. The decoder accepts a compressed bit stream as input and produces a decoded video stream as output. The parser (line 1) performs variable length decoding (VLD) and interprets the bitstream. Because it interprets the bitstream, it dictates decoding parameters which are propagated to the appropriate downstream filters using teleport messaging. The parser outputs macroblocks sequentially, first emitting the blocks contained within a macroblock and then the differentially coded motion vectors associated with the macroblock.

[1]The code and figures throughout this paper are meant to highlight important implementation details. Figures showing every detail of the MPEG decoder and encoder implementations are available on the StreamIt MPEG-2 website [57] and are not reproduced in an appendix because the high level of detail demands an extreme resolution that would be unreadable on regular paper sizes.

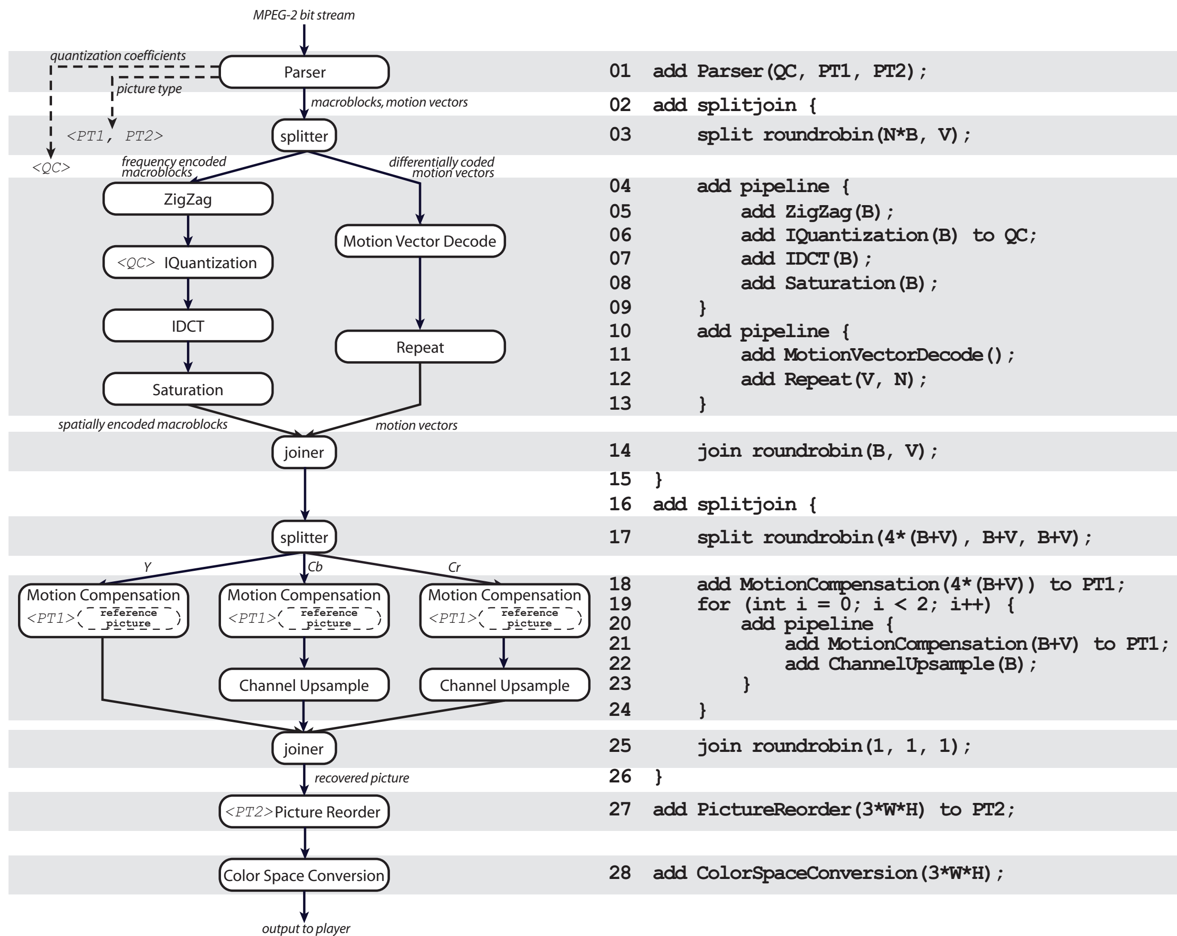


Figure 5-1: MPEG-2 decoder block diagram with associated StreamIt code.

The parser output is segregated into two homogeneous streams by a roundrobin splitter (line 3). The first stream contains blocks, which are spatially decoded by the block decoder (lines 4-9). The second stream is decoded to produce absolute motion vectors (lines 10-13). The two streams are merged with a roundrobin joiner (line 14), which alternates between a block from the left stream, and a set of motion vectors from the right stream.

The next stage of the decoding pipeline performs the motion compensation (lines 16-26) to recover predictively coded macroblocks. Whereas the first half of the pipeline made a split between block data and motion data, the second half of the pipeline splits data according to color channel. The splitter (line 17) first segregates the luminance data into the left stream, and the two sets of chrominance data into

the middle and right streams. The exact amount of data sent each way in this splitter is dependent on the chroma format of the video. The motion compensation filters each take in block data and motion vector data 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. Reference pictures are stored within the decoder for future use.

The two chrominance channels each require upsampling so that every pixel in a video has a value. Finally, the color channels are merged by a joiner (line 25) which merges one item at a time from each stream so that each pixel occurs sequentially.

The final two stages of decoding are picture reordering and color channel conversion. Picture reordering (line 27) ensures that the data is emitted in temporal order, and the color space conversion (line 28) ensures that the data is in the correct output format.

5.2 Encoder Structure

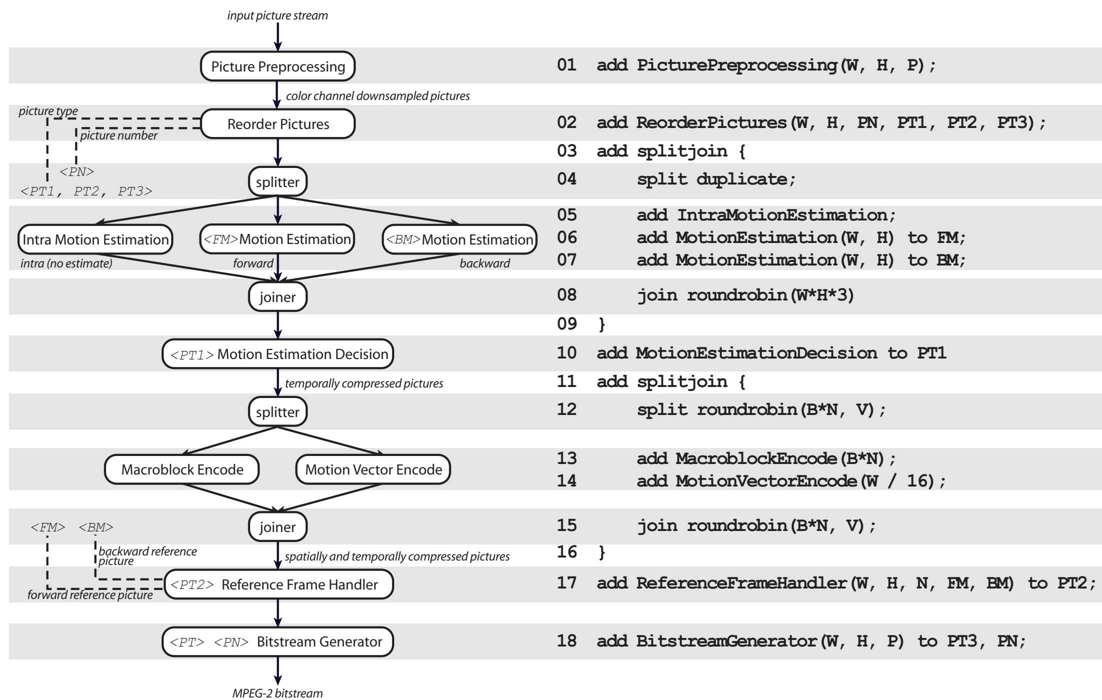


Figure 5-2: MPEG-2 encoder block diagram with associated StreamIt code.

Figure 5-2 shows the MPEG-2 encoder pipeline, correlated with the StreamIt code. Because the MPEG-2 encoder is a larger application and many subcomponents have been explained in the decoder, it is presented at a higher level of abstraction.

The encoder accepts a series of pictures as input and produces a coded bit stream as output. A picture preprocessor (line 1) downsamples the color channels and assigns an I, P, or B picture type to each picture. The output is reordered (line 2) so that all reference pictures precede pictures that use them as references. The component responsible for reordering also sends this information via message to downstream filters which have behavior dependent on the picture type. The output of the picture reorder filter is a sequence of pictures, ordered by macroblock and then block.

The reordered picture output is then motion estimated (lines 3-10) to determine the relative motion of each picture with respect to reference pictures. This component receives upstream messages which tell it what reference frames the decoder will have available, and what they look like. Because the motion estimation has some particularly interesting implementation issues, a more detailed explanation of its behavior follows in Section 5.3.

The motion estimation outputs a series of interleaved blocks and motion vectors. A splitter (line 12) separates these blocks and motion vectors. The blocks are spatially coded (line 13) and the motion vectors differentially coded (line 14) in a computation that is exactly opposite from the spatial and motion vector decoding in the decoder. The data is reinterleaved by a joiner (line 15). The data is now spatially and temporally coded and ready for Huffman compression. However, it is first passed to a reference frame handler (line 17), which is responsible for sending relevant reference frame data back to the motion estimation step. After the reference information is sent upstream, the bitstream generator (line 18) performs Huffman coding on the data and generates an MPEG-2 compliant bitstream as output.

5.3 Motion Estimation

Each macroblock in a picture can be intra coded, forward predicted, or backward predicted. Because all valid forms of encoding must be tried to obtain the best compression, this suggests a stream graph structure in which filters responsible for each encoding type generate candidate blocks in parallel. This graph appears in Figure 5-3. A duplicate splitjoin sends a copy of each picture to each of three filters which produce candidates showing the effects of no prediction, forward prediction, and backward prediction. The outputs of these filters are interleaved by a roundrobin joiner. The interleaved results go to a filter that determines the best encoding technique from the picture type and the compression provided by the different motion compensation strategies. For B pictures, it will try combining the forward and backward estimates together to produce a bidirectional prediction.

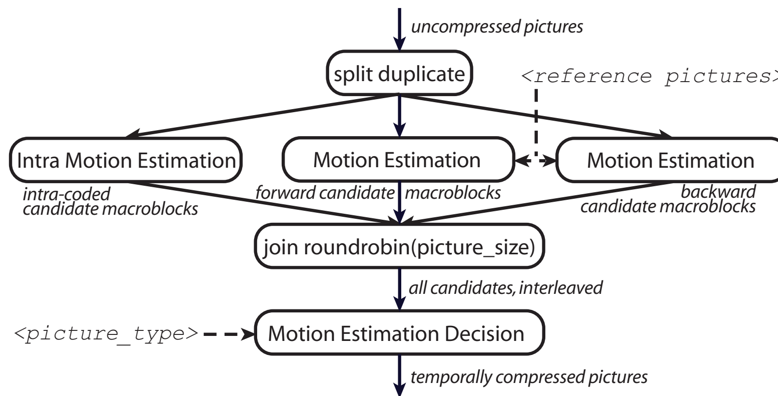


Figure 5-3: Motion estimation stream subgraph.

Each of the two motion estimation filters receives messages containing the previous reference frame to use for estimation. The reference frame is the reference frame *as the decoder would see it* so the actual reference pictures are not determined till near the end of the overall MPEG-2 pipeline. The MPEG-2 encoder thus relies on upstream messages to carry the reference pictures back. Upstream messages are defined with respect to the data being received by the sender, rather than the data being pushed, but otherwise have the same semantics as ordinary messages [63].

	Line Count
Decoder	1357
Encoder	1513
Shared	925

Figure 5-4: StreamIt line counts for decoder-only, encoder-only, and shared library code.

	Filters	Pipelines	Splitjoins
Decoder	6	9	9
Encoder	16	15	18
Shared	17	10	5

Figure 5-5: StreamIt declaration counts for decoder-only, encoder-only, and shared library stream components.

5.4 Implementation Statistics

The MPEG-2 decoder and encoder implementations are fully functional. The codecs support progressive streams which conform to the MPEG-2 Main Profile (See [28], P.106). Video parameters such as chrominance format and video dimensions are currently set in the source code and the application must be recompiled if they change. Code organization and installation instructions can be found at the StreamIt MPEG-2 website [57] and the code is available for download [58]. The decoder implementation required approximately 8 weeks given no prior knowledge of any of the MPEG-2 specifications. (I had prior experience with JPEG and the StreamIt language.) The encoder implementation followed the decoder implementation and required a similar length of time. Line counts[2] for the decoder, encoder, and common functionality between them, are shown in Table 5-4. (All code reused between decoder and encoder appears in the shared library number, not in the decoder or encoder numbers.) The short implementation time and the size of the code base indicates in a rough fashion StreamIt's ability to efficiently express the MPEG-2 computation.

Table 5-5 shows the number of static streams in the MPEG-2 codecs. By static

[2]Line counts were generated using the `SLOCcount` [54] tool. It strips whitespace and comments.

streams, I mean the number of components programmed by myself. At compile-time, due to multiple instantiation and component reuse, the decoder and encoder resolve to stream graphs containing between 600 and 6000 components. The range is due to some parameterization of the parallelism in the stream graph.

Chapter 6

Programmability and Productivity

As previously mentioned, the StreamIt language aims to improve programmability for streaming applications. This chapter expands on this topic by giving specific instances from the MPEG-2 codec implementations where StreamIt improved programmer productivity.

Section 6.1 focusses on StreamIt's implicit buffer management. Section 6.2 describes how pipelines preserve the block diagram structure in the program definition and provide a one-to-one mapping with code. Sections 6.3 and 6.4 show StreamIt's ability to expose data distribution and how this leads to a high degree of malleability. Section 6.5 describes the advantages of hierarchical stream graph construction and Section 6.6 the advantages of teleport messaging.

Because I particularly want to contrast with traditional languages, a number of C code comparisons appear in this and the subsequent section. These code examples come from the C reference decoder implementation [67] provided by the MPEG Software Simulation Group and used in the MediaBench [40] benchmark suite. Because our StreamIt code does not support interlacing or certain optional bitstream semantics, I have modified the C reference implementation to remove that additional functionality, for the purposes of fair comparison. The comparison is between the StreamIt decoder of 2282 lines, and the C reference code of 3477 lines[1]. I believe this line count comparison is a fair quantization of StreamIt's ability to concisely

[1]As before, line counts were generated using `SLOCcount` [54].

express the MPEG-2 decoder computations.

6.1 Buffer Management

Figure 6-1 shows the C code responsible for merging the spatially and temporally decoded block data[2]. Prior to the execution of this function the IDCT function has generated the motion prediction error and the motion compensation function has generated the motion prediction. These two sets of data are summed to produce the decoded blocks.

Lines 9 through 25 determine the appropriate memory addresses for the next block to be processed. Lines 26 to 33 perform the summation. Note that line 29 is the only line actually performing the summation. The buffer management details dominate every line of code and obscure its functional purpose. The buffer management details are particularly complicated because the address of the data is dependent on the block's position in a picture, the size of the video, and the chroma format. The complicated arithmetic used to adjust buffer indices will make it challenging for a compiler to extract parallelism. Further complicating any compiler analysis is the function's reuse of an input buffer as an output buffer (as reflected in line 29).

Reusing buffers is one of many buffer management strategies. The strategy yielding the best performance depends on the target architecture and the size of the buffer. However, the programmer has been forced to guess about the performance and commit to a particular strategy. Also note that a programmer using this function must manually determine an execution schedule and buffer sizes that avoid buffer underflow or the premature overwriting of data.

Now consider the equivalent StreamIt code for adding the decoded block data, shown in Figure 6-2. The code itself is almost trivial. The filter occurs after a `roundrobin(1)` joiner merging the prediction error and the prediction itself. Fig-

[2]The original Add_Block function performed some unrelated parts of the decoding process as well and was almost 99 lines long. For comparison purposes the unrelated code is removed.

```
01 static void Add_Block(comp,bx,by)
02      int comp,bx,by;
03 {
04   int cc,i, j, iincr;
05   unsigned char *rfp;
06   short *bp;
08   cc = (comp<4) ? 0 : (comp&1)+1;
09   if (cc==0) {
10     rfp = current_frame[0] +
11           Coded_Picture_Width*(by+((comp&2)<<2)) +
12           bx +
13           ((comp&1)<<3);
14     iincr = Coded_Picture_Width - 8;
15     if (chroma_format!=CHROMA444)
16       bx >>= 1;
17     if (chroma_format==CHROMA420)
18       by >>= 1;
19     rfp = current_frame[cc] +
20           Chroma_Width*(by+((comp&2)<<2)) +
21           bx +
22           (comp&8);
23     iincr = Chroma_Width - 8;
25   }
26   bp = ld->block[comp];
27   for (i=0; i<8; i++) {
28     for (j=0; j<8; j++) {
29       *rfp = *bp++ + *rfp;
30       rfp++;
31     }
32     rfp+= iincr;
33   }
34 }
```

Figure 6-1: Combining the spatially and temporally decoded data in C.

```
01 int->int filter Add_Block {
02   work pop 2 push 1 {
03     push(pop()+pop());
04   }
05 }
```

Figure 6-2: Combining the spatially and temporally decoded data in StreamIt.

ure 7-4, explained in detail later in Chapter 7, shows this filter in the context of the motion compensation stream graph. Note that the amount of data this filter processes will be dependent on the same chrominance and picture size parameters as the C code but the buffer management details are hidden from the programmer - they will be reflected in the data rates of the splitters and joiners surrounding the filter in the motion compensation graph. At compile time the compiler makes the decisions about the best buffer management strategy [52].

6.2 Pipelines Preserve Block Structure

The pipeline construct preserves the structure implicit in a block diagram. Figure 6-3 shows a block diagram for spatial decoding taken from Figures 7-1 and 7-4 of the MPEG-2 specification [28]. Figure 6-4 shows the StreamIt code that implements the pipeline. The obvious correspondence points to StreamIt's ability to naturally represent this computation. Note that the quantization parameters in the diagram are realized as messages in the stream graph.

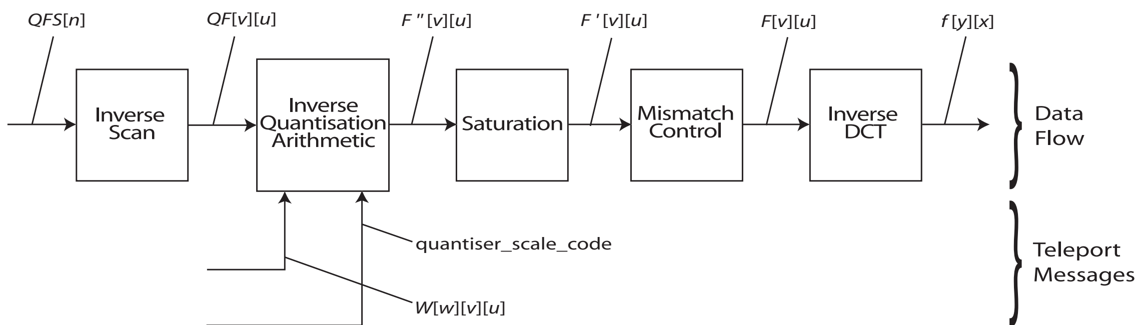


Figure 6-3: Block diagram for spatial decoding (from MPEG-2 specification).

An important detail to note is that each of the filters used in a pipeline may have different granularities. In this case the inverse scan, quantization, and IDCT all operate on blocks of 64 data values at a time. The mismatch control and saturation blocks are naturally expressed in terms of a single input and output token. Nowhere must the programmer specify the number of executions or the execution sequence needed to fully decode a picture or video. This granularity variance holds through

```
int->int pipeline BlockDecode(
  portal<InverseQuantization> quantiserData,
  portal<MacroblockType> macroblockType) {

  int[64] Order = {...};
  add ZigZag(64, Order);
  add InverseQuantization() to quantiserData,
                               macroblockType;
  add Saturation(-2048, 2047);
  add MismatchControl();
  add 2D_iDCT(8); // 8x8 2D IDCT
  add Saturation(-256, 255);

}
```

Figure 6-4: StreamIt pipeline for spatial decoding.

the rest of MPEG-2 as well. Picture reordering can be expressed in terms of pictures and motion compensation in terms of motion vectors. The programmer's burden is eased since he does not have to worry about the rate discrepancies between filters.

6.3 Natural Exposure of Data Distribution

The splitjoin construct is a robust mechanism for expressing parallelism and data reordering operations. It can be used to specify coarse grained parallelism at the highest levels of an application because it exposes the independence of computational blocks. But it is also useful for describing how a computation should be performed, exposing coarse grained and fine grained parallelism as a byproduct.

Channel upsampling illustrates this. One can easily think of 2D upsampling in terms of 1D upsampling in the vertical and horizontal direction[3]. The natural way to express upsampling a picture in either direction is as a parallel computation on every row or column of the picture. The building block is a one-dimensional upsampling filter that takes in a pair of weights used for interpolation between points.

[3]This is conceptually similar to the decomposition of a 2D DCT, discussed later in Section 7.2. In this case the channel upsampling represents a smaller amount of work and the splitjoin is more interesting for its ability to efficiently express the transformation than its ability to expose parallelism.

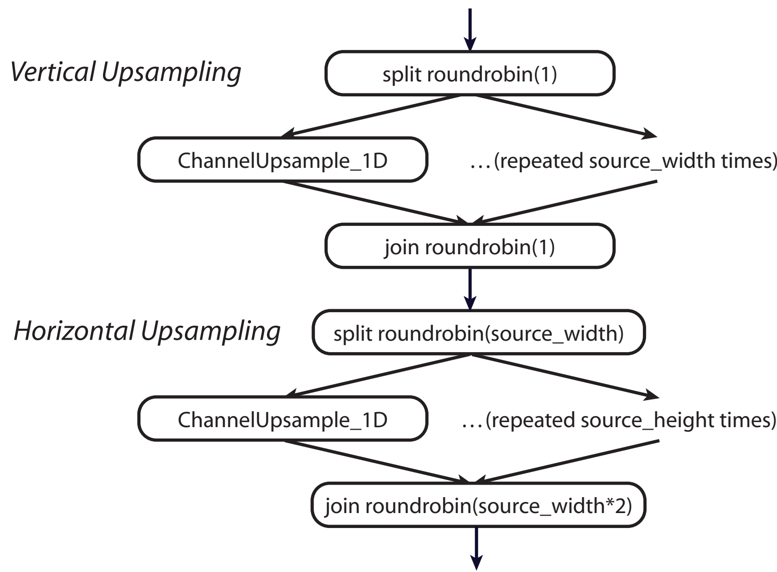


Figure 6-5: 2D upsampling decomposed into 1D upsampling

```
int->int splitjoin ChannelUpsample_Vertical(int sourcewidth,
                                            int sourceheight) {
    split roundrobin(1);
    for (int i = 0; i < sourcewidth; i++) {
        add ChannelUpsample_1D(sourceheight, 0.75, 0.25);
    }
    join roundrobin(1);
}

int->int splitjoin ChannelUpsample_Horizontal(int sourcewidth,
                                              int sourceheight) {
    split roundrobin(sourcewidth);
    for (int i = 0; i < sourceheight; i++) {
        add ChannelUpsample_1D(sourcewidth, 0.5, 0.5);
    }
    join roundrobin(sourcewidth*2);
}
```

Figure 6-6: Splitjoins for channel upsampling.

```
   /* Y */
01 form_component_prediction(src[0]+(sfield?lx2>>1:0),
02                           dst[0]+(dfield?lx2>>1:0),
03                           lx,lx2,w,h,x,y,dx,dy,average_flag);
04 if (chroma_format!=CHROMA444)  {
05    lx>>=1; lx2>>=1; w>>=1; x>>=1; dx/=2;
06 }
07 if (chroma_format==CHROMA420)  {
08   h>>=1; y>>=1; dy/=2;
09 }
   /* Cb */
10 form_component_prediction(src[1]+(sfield?lx2>>1:0),
11                           dst[1]+(dfield?lx2>>1:0),
12                           lx,lx2,w,h,x,y,dx,dy,average_flag);
   /* Cr */
13 form_component_prediction(src[2]+(sfield?lx2>>1:0),
14                           dst[2]+(dfield?lx2>>1:0),
15                           lx,lx2,w,h,x,y,dx,dy,average_flag);
```

Figure 6-7: C code excerpt for handling 4:2:0 and 4:2:2 chroma formats.

The weights are different for horizontal and vertical upsampling because MPEG-2 downsamples horizontally by removing pixels, and vertically by displacing them. The vertical upsampler splits the data by column and the horizontal upsampler splits the data by row. Figure 6-5 illustrates the procedure and Figure 6-6 shows the StreamIt code that implements the process. The C reference implementation implements upsampling as a series of loops wrapped around a one dimensional upsampling kernel. In this case the code looks similar to the StreamIt code. However, the StreamIt `for` loops represent graph topology resolved at initialization time. The C compiler must analyze the body of the loops to extract parallelism.

6.4 Code Malleability

A noteworthy aspect of the StreamIt implementation is its malleability. I illustrate this by outlining how the decoder implementation is modified to support both 4:2:0 and 4:2:2 chroma formats (see Section 2.2). The conceptual difference between chroma formats is merely a change in downsampling ratio. The implementation

```
// B = amount of data per block
// V = amount of data per motion vector
add splitjoin {
  split roundrobin(4*(B+V), B+V, B+V);
  add MotionCompensation(4*(B+V)) to PT1;
  for (int i = 0; i < 2; i++) {
    add pipeline {
      add MotionCompensation(B+V) to PT1;
      add ChannelUpsample(B);
    }
  }
  join roundrobin(1, 1, 1);
}
```

Figure 6-8: Original StreamIt code excerpt for handling 4:2:0 chroma format only.

```
// C = blocks per chroma channel
//     per macroblock
// C = 1 for 4:2:0, C = 2 for 4:2:2
// B = amount of data per block
// V = amount of data per motion vector
add splitjoin {
  split roundrobin(4*(B+V), 2*C*(B+V));
  add MotionCompensation(4*(B+V)) to PT1;
  add splitjoin {
    split roundrobin(B+V, B+V);
    for (int i = 0; i < 2; i++) {
      add pipeline {
        add MotionCompensation(B+V) to PT1;
        add ChannelUpsample(C*B);
      }
    }
    join roundrobin(1, 1);
  }
  join roundrobin(1, 1, 1);
}
```

Figure 6-9: StreamIt code excerpt for handling 4:2:0 and 4:2:2 chroma formats.

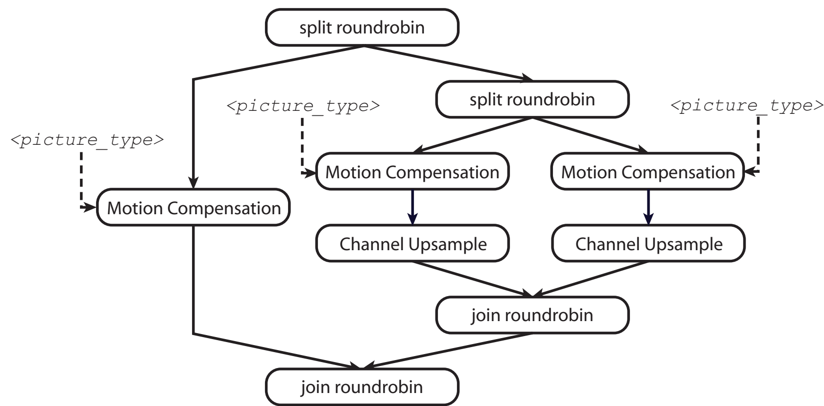


Figure 6-10: Modified subgraph for handling 4:2:0 and 4:2:2 chroma formats.

difference is that the 4:2:0 format represents a macroblock with six blocks, and the 4:2:2 with 8 blocks. This change affects the data rates, and the data splitting ratios 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 flag** to dictate control flow and alternate index/offset calculations in 43 locations in the code. As an example, Figure 6-7 shows a code fragment from the `form_prediction` routine in `recon.c`. 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, I 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 17 of Figure 5-1. 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, as previously shown in Figure 2-3. Thus, a nested splitjoin is used to properly recover the chrominance channels.

The code for the old splitjoin before the chroma format support modifications is shown in Figure 6-8. The code for the new splitjoin is shown in Figure 6-9 and illustrated in Figure 6-10. In the StreamIt code, the chroma format explicitly dictates control flow in only 9 locations. Of course, chroma format changes have effects on scheduling and buffer management, but this is transparent to the programmer.

6.5 Hierarchical Construction vs Functional Calls

Preserving the block structure in the program definition is important for programmer productivity and maintaining code malleability. Figure 6-11 shows what happens to the block diagram in a traditional language. This figure represents a simplified call trace for the MPEG-2 decoder in C. Note that each component must be wrapped in loops determining the exact number of iterations the component must run to fully decode a picture or video. Data is stored in global buffers and addresses to these buffers are passed between functions, so the code structure fails to reflect the actual movement of data. Finally, notice that the functions for parsing, motion compensation, and spatial decoding are intermixed in the code for performance reasons. The StreamIt compiler can interleave the phases of execution to provide performance but this does not require the programmer to mix functional code.

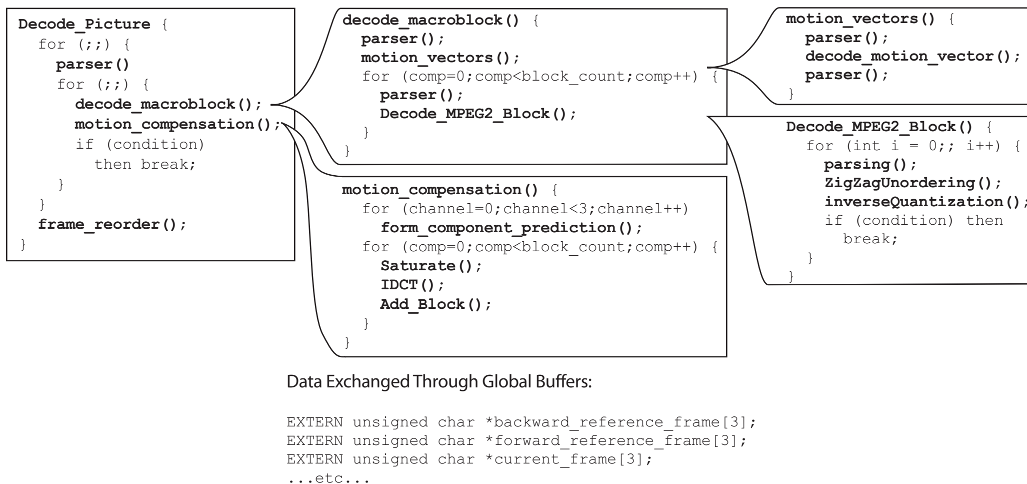


Figure 6-11: Simplified call-trace diagram for C decoder.

6.6 Usefulness of Teleport Messaging

Teleport messaging is a useful language construct, as demonstrated by its importance in handling control messages in the decoder and encoder. Figures 6-12 and 6-13 show the important control parameters that are sent by teleport messages in the MPEG-2 codecs. The number of subscribers for each message type highlights the usefulness of messages.

Message Content	**Frequency**	**Regular**	**Number of Subscribers**	**Number of Subscriber Types**
quantization scale code	macroblock	no	2	2
quantization tables	video	yes	2	2
macroblock type	macroblock	yes	4	2
picture type	picture	yes	4	2
motion vector resets	macroblock	no	1	1
reference pictures	picture	no	many [4]	1

[4] One for every block in all color channels over a picture.

Figure 6-12: Important control parameters sent through the decoder using teleport messaging.

Message Content	**Frequency**	**Regular**	**Number of Subscribers**	**Number of Subscriber Types**
quantization scale code	macroblock	no	2	2
quantization tables	video	yes	1	1
macroblock type	macroblock	yes	4	2
picture type	picture	yes	4	4
reference pictures	picture	no	many [5]	2

[5] Once for every macroblock in all color channels over a picture.

Figure 6-13: Important control parameters sent through the encoder using teleport messaging.

Messaging's impact on programmability is evident by considering how the C code exposes control relevant information. The C code passes data and control parameters through function parameters and a global address space. Figure 6-14

shows the functional units in the C decoder[6], the input bitstream, output video, and the shared address space. Arrows represent communication dependencies between the blocks. While the C implementation is natural — for C code — it would be difficult for a programmer or compiler to extract parallelism[7].

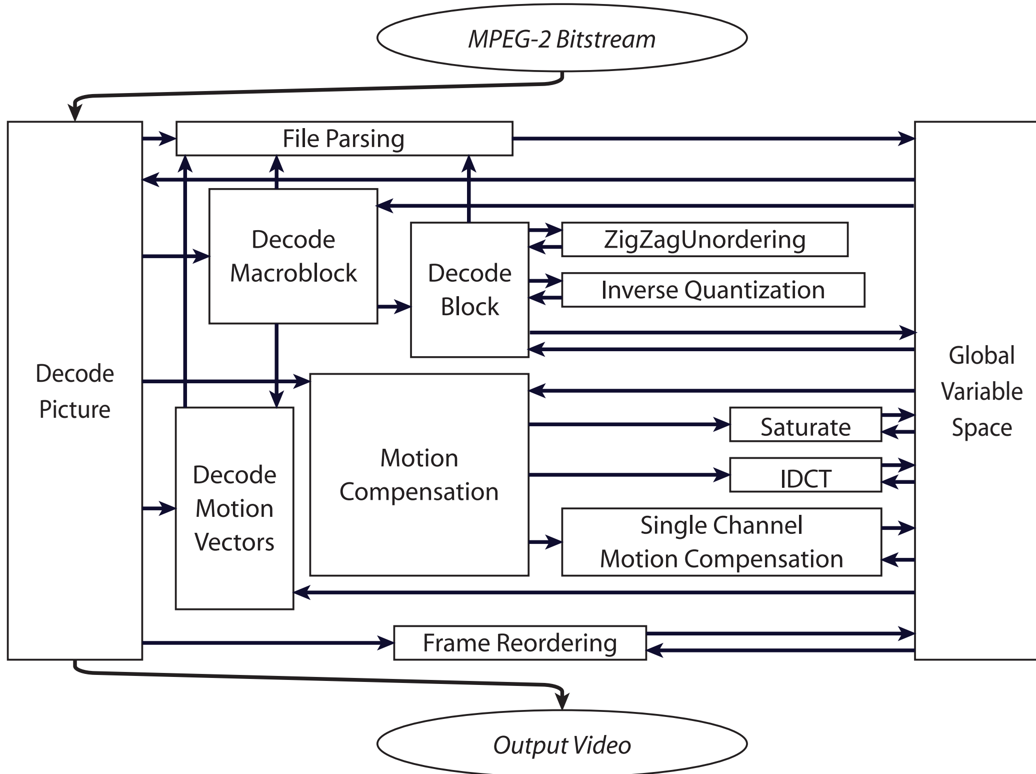


Figure 6-14: Communication dependencies between functional units in the C code.

Teleport messaging provides a point to point connection layer which a programmer can emulate in the dataflow layer by embedding control parameters in the data channels. While StreamIt provides the feedback loop construct to send information upstream, messaging is often more appropriate for altering upstream state.

Upstream reference frame passing in the encoder is used as an example. A feedback loop structure is impossible here, due to the fact that only some pictures should be sent upstream as reference pictures, and the information about which pictures to send upstream is not available at compile time. This would require the feedback loop

[6]A functional unit might contain multiple functions with similar or related behaviors.

[7]Because the analysis of data access patterns is complicated and correctness verification for optimizations becomes impractical, a compiler will make a pessimistic decision.

to process messages regarding picture type and implement a dynamic rate splitter and joiner. This is not intuitive to implement, nor would it be easy for the compiler to analyze. Without B pictures the feedback loop structure needed to ensure that each motion estimation filter received the right reference frame immediately before the first block of a picture would still be non-trivial. Replacing the messaging scheme with a feedback loop would also effectively require the motion estimation filters to operate at a picture granularity, since each work execution would receive a reference picture, and this would limit parallelism by preventing estimation to be expressed at a block granularity.

This chapter has given specific instances of where StreamIt improved programmer productivity. Personal qualitative observations on the entire implementation process are also pertinent. I found programming in StreamIt to be very intuitive. MPEG-2 decomposed naturally into concise independent filters. Filters make a program more modular than equivalent C code, which made it easy to change the encoder and decoder stream graphs as my understanding of MPEG-2 improved. For the purposes of correctness verification and performance comparisons it was much easier to extract functional units from the StreamIt code than the C code.

Chapter 7

Expressing Parallelism

Multimedia codecs such as MPEG-2 demand high performance because of throughput requirements. An online encoder or decoder must be capable of realtime performance, which can be as high as 60 fps. An offline encoder is less constrained but minimizing total runtime is still a concern. Since traditional CPU clock scaling has ended, meeting throughput demands requires parallel scalable implementations. Critical to these implementations is the compiler's ability to detect parallelism in a program specification. This section examines how parallelism was successfully realized in the MPEG-2 codecs and how the StreamIt language made this task easy. The parallelism discussed in this section is exposed through the stream graph topology rather than by changing any underlying algorithms. This section ignores parallelizing MPEG-2 codecs across GOPs; since GOPs are coded independently, this is embarrassingly parallel and trivial. I also include proof of concept results showing that the StreamIt implementation scales on multicore architectures.

7.1 Splitjoins Express Data Parallelism

One particularly noteworthy aspect of splitjoins is the ability to define their internal topology using the for-loop construct. The for-loop, unrolled at instantiation time, makes the degree of parallelism and the stream topology itself parameterizable. This feature makes it easy for the programmer to concisely express a data parallel

computation. The code responsible for channel upsampling, shown in Figure 6-6, expresses a massively data parallel computation in this way.

7.2 Hierarchical Streams Expose High Degrees of Parallelism

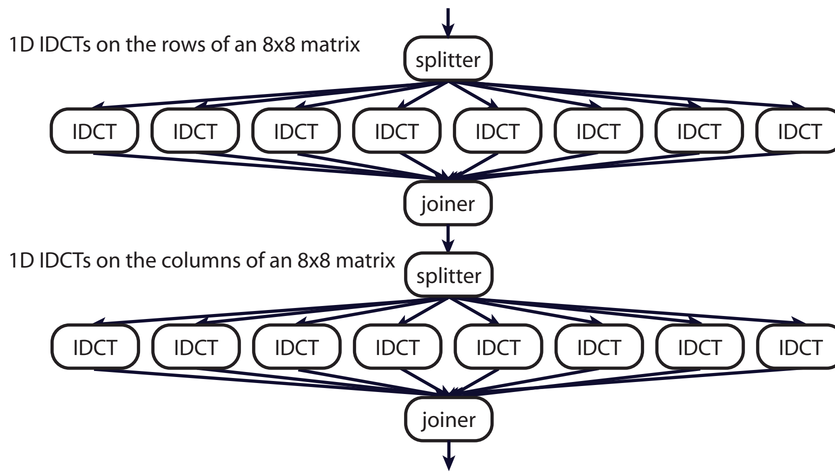


Figure 7-1: Subgraph for a fine grained 2D inverse DCT.

Hierarchical constructs provide a convenient and natural way to represent parallel computation. Figure 7-1 shows a parallel implementation of the 2D IDCT using 1D IDCTs. This implementation is both data parallel (within the rows and columns) and pipeline parallel (between the rows and columns). The StreamIt code for this 2D IDCT appears in Figure 7-2. A straightforward C implementation of a computationally equivalent IDCT is shown in Figure 7-3. Note that the code structure is similar to the StreamIt version, although it does not explicitly expose the parallelism; the compiler must perform loop and dependency analysis to enable 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

```
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 7-2: StreamIt code for the fine grained 2D inverse DCT subgraph.

buffer management [52]. The StreamIt implementation is also parameterized such that it is trivial to adjust the size of the IDCT.

7.3 Parallelizing Motion Prediction

Motion prediction, in both the decoder and encoder, represents a significant fraction of the computational effort, and is amenable to data parallelism. Because each block in a picture has an associated set of motion vectors, motion prediction filters

```
// 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 7-3: C code for 2D inverse DCT calculation using two 1D transforms.

express a block-level transformation, and predictions for blocks may be formed in parallel. However, the act of forming the prediction requires a filter have access to full reference pictures. A parallel implementation of motion prediction will either lead to redundant copies of the reference pictures or the necessity for motion prediction threads to share a global read only memory space where the reference pictures can be stored. A parallel motion prediction algorithm is easy to express in StreamIt and exposes the necessary information so that the compiler can provide a shared-memory storage for the reference pictures.

Figure 7-4 shows the StreamIt pipeline responsible for motion compensation of a single color channel in the decoder. All blocks in a given picture are motion predicted in parallel, with a set of block prediction error values and motion vector

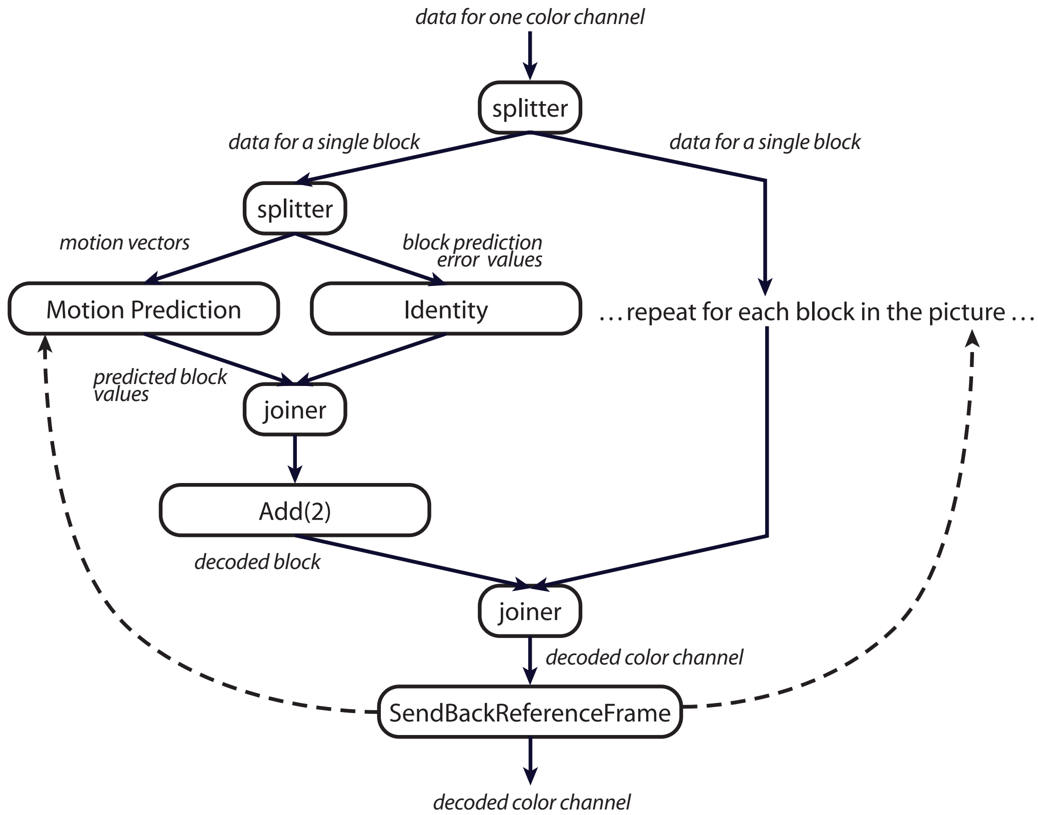


Figure 7-4: Stream topology for motion compensation of a single color channel.

information sent to each motion prediction filter. Each data set then has its block coefficients (which need no processing) and the motion vectors splitup, sending the motion vectors to the filter that actually does the work of computing the predicted block using the motion vector offsets. The predicted block output is added to the block prediction error to form the original block, and all blocks are interleaved in the data streams. The final filter in the stream graph is responsible for communicating this fully decoded picture back to the motion prediction filters. It does this by sending upstream messages containing the reference picture data.

Note the cleanliness of this parallel implementation: a programmer could manually change the number of blocks decoded in parallel and similarly, the compiler can automatically fuse filter instances to match the target hardware. Note also that the reference pictures needed by the motion compensation filter and the picture type data needed by both the motion compensation and reference frame sending filters

are naturally expressed using messaging. Finally, note that the motion compensation filters only need to form predictions using their available reference pictures and a separate filter can make sure they access the correct references.

As mentioned previously, the one potential drawback of this scheme would be if each instance of the motion prediction filter required its own instance of the reference picture data. However, a simple optimization allows the compiler to detect that these motion prediction filters only use these reference pictures in a read-only context and they may be stored in a global address space. Suppose that a filter receives a message containing a nugget of data X. If a compiler can detect that the filter never modifies X, it can place a copy in a shared memory space and give the filter a reference to X. Whenever a new message is received, it simply gives the filter a new reference to the data in a different part of the shared memory space. If a filter modifies X or the compiler cannot detect that the filter uses X immutably, it must give the filter a copy of the message in its own address space. Some kind of reference count or garbage collection mechanism must be associated with data stored in the global space so that the memory can be freed or reused once no filters can access the data.

The read-only analysis and associated space saving optimization can be performed by the compiler automatically. This enables program implementations to use a global reference space, even though the language need not provide this feature. I have tested a prototype of this optimization in the StreamIt Java library, where garbage collection is easily handled, and verified that it works.

7.4 Improving Decoder Parallelization

Most MPEG-2 decoder implementations make spatial decoding precede temporal decoding, but nothing in the specification or decoding algorithms require this. To achieve a higher degree of parallelization, one would want to generate the predictions for a block in parallel with the spatial interpretation of the block coefficients. This has been tried with success in a hardware based implementation [51]. Only near the final output stage must the prediction and spatial data be summed and the output

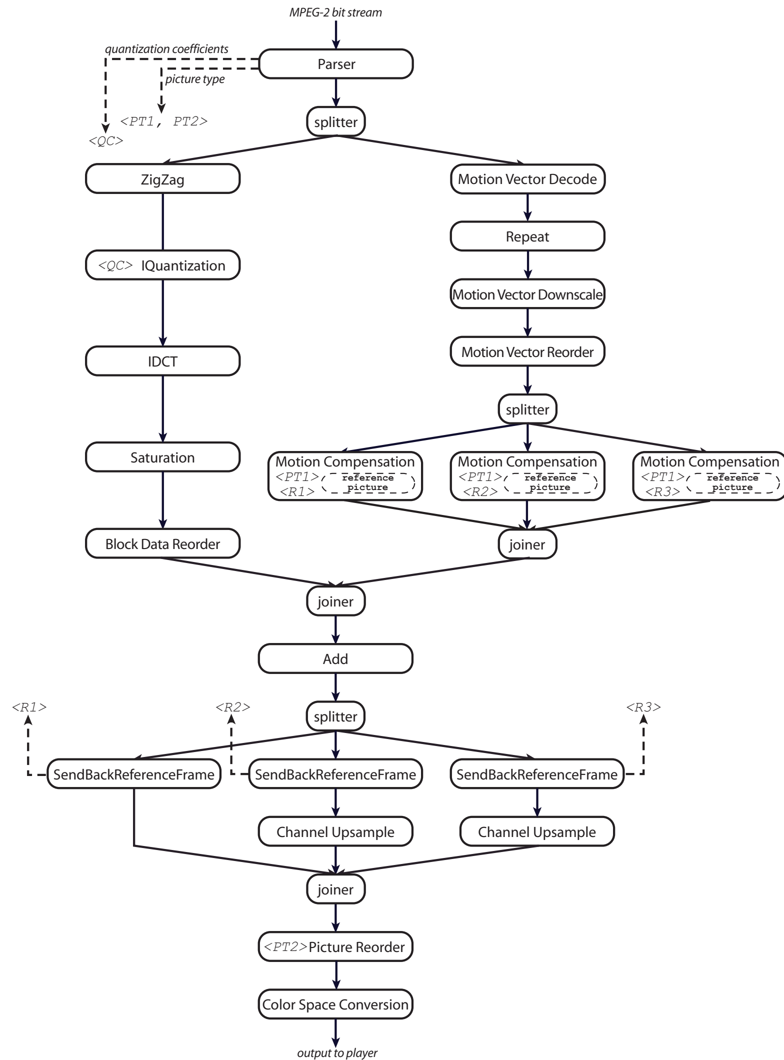


Figure 7-5: Exposing parallelism between spatial and temporal decoding in an MPEG-2 decoder.

generated.

I previously showed a block diagram for the MPEG-2 decoder in StreamIt (in Figure 5-1). This diagram represents both a straightforward block specification and a program definition derived from the specification. Figure 7-5 shows an alternative block diagram for an MPEG-2 decoder. This alternative diagram shows a block diagram that an MPEG-2 expert will decide upon[1]. These two block diagrams

[1]The original diagram represents the program structure I chose a month after originally familiarizing myself with the MPEG-2 specification. The new diagram represents the program structure I chose 8 months later after extensive familiarity with the specifications and implementations.

contain the same functional transformations. For an ideal language, a clean and malleable implementation of the original block specification in Figure 5-1 could easily be modified to expose the greater degree of parallelism present in the expert's block structure in Figure 7-5.

Because StreamIt preserves block structure in the program definition this transformation is easily accomplished. I have implemented the alternate decoder, using exactly the same filters as those used in the original decoder implementation. These two implementations are functionally equivalent, generating the same output across test cases. The only changes in the code are to the nested hierarchical container topologies that dictate the decoder stream graph. The ease with which a programmer can expose greater parallelism by merely changing the stream topology points to the usefulness of stream based languages for parallel program implementations.

7.5 Performance Results

This section provides performance results showing how StreamIt enables a compiler to provide scalable performance on MPEG-2 codec implementations. A need for new language features previously mentioned in this thesis prevents the compilation of a full decoder or encoder[2]. Execution and evaluation of the complete MPEG-2 decoder and encoder stream graphs is a current focus of the StreamIt group. Here I consider the spatial decoding subgraph, which represents up to a third of the total computation in the decoder [31]. I have extracted the spatial decoding code from the StreamIt and C decoder implementations so that it can be compiled and run on a performance oriented target.

I use three versions of the StreamIt spatial decoding stream graph with differing granularities. The granularities are changed by altering the subgraph for the 8x8 IDCT. Figure 7-6 lists the number of nodes (filters, splitters, and joiners), contained in the stream graph for each of the three versions. Each version contains

[2] An intermediate Java library output from the compiler allows me to verify correctness of StreamIt code. This target is not performance oriented.

Granularity	Nodes in IDCT	Nodes in Graph
Coarse	2	8
Intermediate	11	17
Fine	20	26

Figure 7-6: Three versions of the spatial decoding stream graph and their granularities.

`Source`, `ZigZag`, `InverseQuantization`, `Saturation`, `MismatchControl`, and `Sink` filters. The coarse grained IDCT uses 2 filters for the IDCT: a `Rows_iDCT` and a `Columns_iDCT` filter. The fine grained IDCT is expressed with 16 1D IDCTs and two splitjoins as described in Section 7.2. The intermediate IDCT uses the coarse `Rows_iDCT` filter for the row transformation and the fine grained column transformation with 8 1D IDCTs.

The performance evaluation is carried out on the Raw architecture [60, 61]. The Raw architecture is representative of the industry shift to multicore embedded architectures, currently manifested in emerging architectures such as the Intel Duo, AMD Opteron, and IBM Cell processors. Raw is a wire-exposed multicore architecture which contains a 2D array of identical, programmable tiles and supports instruction, data, thread, and pipeline parallelism. Each tile has a compute processor and a switch processor that manages communication. The compute processor is composed of an eight-stage in-order single-issue MIPS-style processor, a four-stage pipelined floating point unit, a 32kB data cache, and a 32kB instruction cache. Tiles are connected by a FIFO queue with a 3 cycle near neighbor latency. This interconnect network provides a mechanism for filters to communicate quickly with each other. The current Raw prototype is a chip with 16 tiles running at 425Mhz. For this thesis, results are gathered using a cycle-accurate simulator [61] that can model tile configurations with size 1, 4 (2x2 grid), and 16 (4x4 grid).

The StreamIt code was compiled to Raw with the StreamIt space multiplexing compiler [3] described in [25]. The code was compiled at an `-O1` optimization level, which performs loop unrolling by a factor of 16, scalar replacement, and aggressive

[3] This compiler will soon be deprecated in favor of a more advanced space-time multiplexing compiler described in an upcoming paper [24] from the StreamIt group.

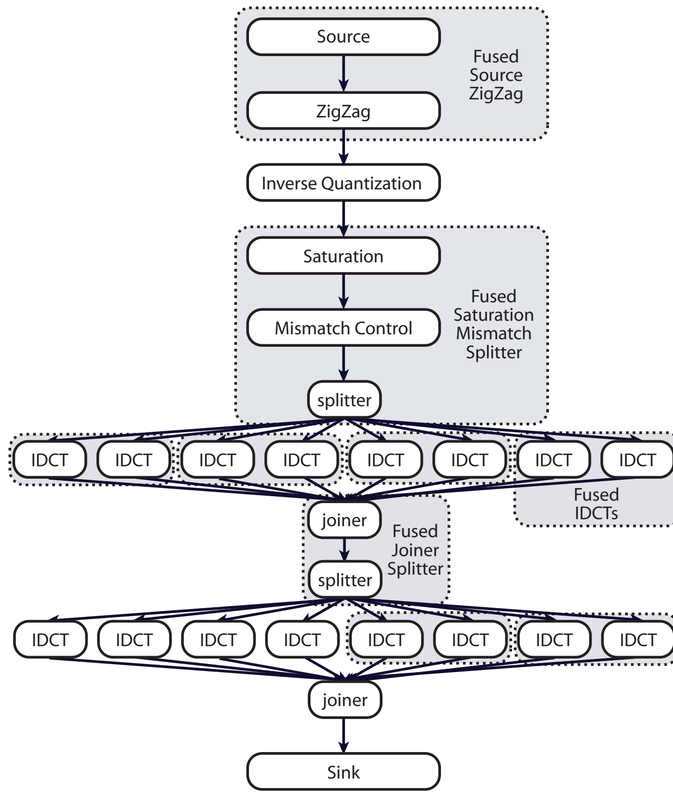


Figure 7-7: Partitioning the spatial decoding stream graph for 16 tiles of Raw.

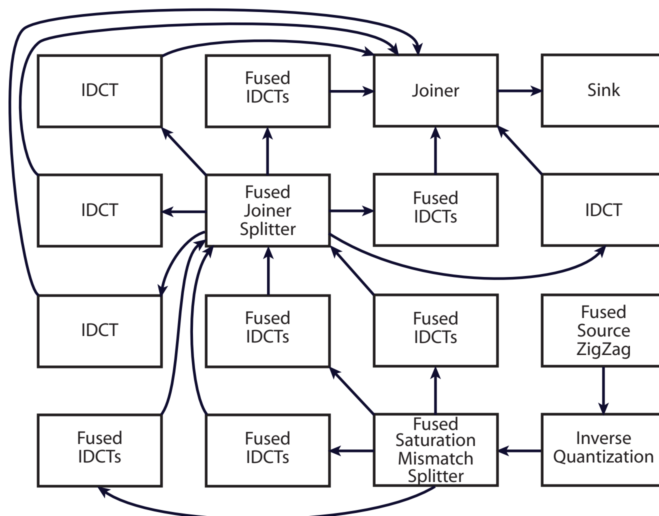


Figure 7-8: Layout of the fine grained spatial decoding stream graph on the Raw chip.

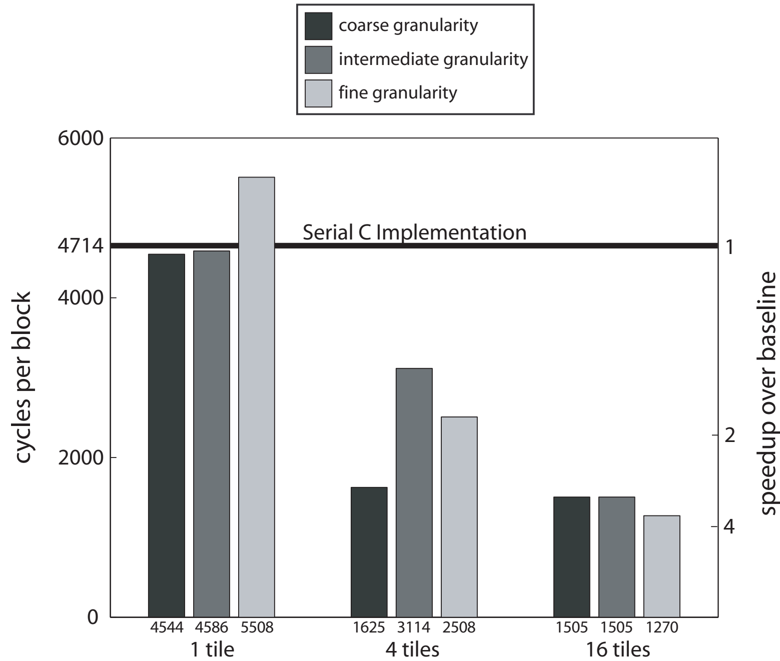


Figure 7-9: Scalability of StreamIt spatial decoding pipeline against single tile C baseline.

constant propagation. The StreamIt compiler focusses on achieving performance by enabling scalable execution on many tiles. Using the parallelism explicit in the stream graph the compiler automatically performs task partitioning and layout. An application should achieve significant speedups as the number of tiles increases.

A brief example illustrating scheduling and layout follows. Figure 7-7 shows how the StreamIt compiler partitions the fine grained spatial decoding stream graph for a Raw configuration with 16 tiles. The compiler adjusts the stream graph granularity by combining nodes to eliminate buffering between nodes on the same tile. The dotted lines in the figure show which nodes in the stream graph have been fused. After fusion is complete there are the same number of nodes and tiles in the stream graph. Figure 7-8 shows the compiler's layout decision for these nodes on the Raw hardware.

Figure 7-9 shows results for execution of the spatial decoding stream graph at three granularities. The baseline is the C reference code running on a single tile. The C code was compiled with a Raw port of the `gcc` compiler, which performs register

allocation and list scheduling. A compiler has a difficult time extracting parallelism from single threaded C code (hence the need for new programming models); in StreamIt the parallelism is explicit, so we show performance numbers for multi-tile configurations as well.

Single tile StreamIt performance is roughly equivalent to the single tile C code. Small variations are explained by compiler decisions about how to fuse all the filters together. When the tile size increases to 4, all versions of the StreamIt code significantly outperform the C implementation. While the coarse grained version of the StreamIt code outperforms the more fine grained implementations, an analysis of the compiler's layout decisions suggests that this result is somewhat anomalous: the compiler is getting exceptionally lucky with load balancing. The intermediate and fine grained versions are more prototypical. At 16 tiles performance of all stream graphs again improves significantly. One sees the general trend that decreasing the granularity of the program enables the compiler to make better layout decisions and provide higher parallel performance. The best performance comes from a fine-grained implementation operating on 16 tiles.

Chapter 8

StreamIt Limitations and Proposed Extensions

During the implementation process I encountered situations where StreamIt forced a poor expression of a computation or language features did not scale for large application development. A language design oriented to small applications and benchmarks could easily miss these pragmatic development issues. The StreamIt group has focussed on these issues; the next version of StreamIt will add a number of language extensions that improve programmability, modularity, and expression of parallelism and communication. This chapter provides concrete examples motivating the introduction of these features and attempts to steer the direction of future streaming languages.

8.1 Bitstream Parsing

In MPEG-2, the process of parsing the bitstream and performing Huffman and run-length coding is estimated to constitute almost a third of the computational effort [31]. Unfortunately, many layers of nested control flow make the parser unsuitable for streaming computation. The parser must be implemented as a single filter that contains approximately one thousand lines of code, almost half the code lines in either the decoder or encoder. This filter expresses scheduling information

and parallelism poorly.

Filters with dynamic IO rates present particular difficulties for the StreamIt compiler because of the lack of scheduling information. However, certain filters are naturally expressed by the programmer with dynamic input and output rates, although they theoretically could be realized with static rates. In the decoder, the parser has a dynamic input rate, but always outputs full macroblocks, and ought to be expressed with a static output rate. (The reverse is true for the parser in the encoder.) However, due to the complexity of the MPEG-2 bitstream format, the actual push statements are embedded in deeply nested control flow. The parser is most naturally expressed in terms of a single work function execution that parses the entire MPEG-2 bitstream.

This presents several difficulties. The fact that the parser processes all its data in a single pass means that if the filter were to execute atomically with respect to other filters, one would need a potentially infinite buffer between the parser and the downstream graph. To avoid this difficulty the compiler would need to generate multi-threaded code, even for a uniprocessor backend, that interleaves the parser execution with downstream execution. Messaging presents a second difficulty. A filter's messaging granularity is determined by its input and output rates, and messages are timed with respect to the work execution boundaries; a work function that processed an entire video in a single execution could only time messages with respect to the beginning or end of the video, not with respect to individual macroblocks within the video.

I present a workaround that allows for interleaved scheduling and message delivery. Given a filter with a large work function, such as the parser, one can move control flow determining variables into filter instance variables that are maintained between work function executions. Inside the work function invocation is a loop that repeatedly branches to one of several helper functions determined by the control variables. The helper functions update the control variables, and the loop terminates after executing any of the helper functions that generates output (for the parser, pushing a macroblock). This workaround amounts to replacing nested control flow

with a flat control flow graph structure. The readability and malleability of the code suffers.

Even with such a hack, parallelism is expressed poorly because the parser itself cannot be parallelized easily. Two constructs in the MPEG-2 bitstream facilitate parser parallelism. At the higher level, every GOP can be decoded in parallel, since pictures can only reference other pictures in the same GOP. At a lower level slices may be decoded in parallel since macroblocks contained inside different slices in a picture are known to have no interdependencies. The codes for the start of a GOP or slice are byte-aligned and never occur in any other context. A parser exploiting thread-level parallelism could first scan through a bitstream and identify the GOP and slice structures, and then subdivide portions of the bitstream to different bitstream parsers. Ahmad et al. show this technique to be effective for parallelizing MPEG-2 [2]. This sort of parser parallelization is difficult to express in a stream based language and achieving reasonable performance would require a Herculean compiler effort.

The StreamIt group has considered language features to address these problems, but our key insight is the realization that StreamIt targets streaming computations, and if a filter cannot concisely express a computation then that computation is probably not streaming. For the MPEG-2 parser this is obvious because its expression requires a thousand line functional block and leaves the compiler with an intractable parallelism problem. MPEG-2 bitstream parsing has little in common with stream computation and much in common with context-free grammars, and would be implemented more easily in a language like C. Clean interfaces between StreamIt and traditional languages, under development in the StreamIt language group, would allow hybrid language compression scheme implementations that share the benefits of both languages. MPEG-2 provides a strong motivation for such a hybrid; while the majority of MPEG-2 coding is particularly well suited to a streaming approach, bitstream parsing is an important exception that demands an alternate approach.

```
...
int horizontal_size_value =
  popbits(12);



int vertical_size_value =
  popbits(12);



int aspect_ratio_information =
  popbits(4);



int frame_rate_code =
  popbits(4);



...
```

```
...
int horizontal_size_value = 0;
for (int i = 0; i < 12; i++) {
  horizontal_size_value <<= 1;
  horizontal_size_value += pop();
}
int vertical_size_value = 0;
for (int i = 0; i < 12; i++) {
  vertical_size_value <<= 1;
  vertical_size_value += pop();
}
int aspect_ratio_information = 0;
for (int i = 0; i < 4; i++) {
  aspect_ratio_information += pop();
  aspect_ratio_information <<= 1;
}
int frame_rate_code = 0;
for (int i = 0; i < 4; i++) {
  frame_rate_code <<= 1;
  frame_rate_code += pop();
}
...
```

Figure 8-1: Code fragment from parser with (left) and without (right) tape accessing external functions.

8.2 Functions with Tape Access

Functions provide an abstraction that improves programmability when a computation must be repeated within a single work execution of a filter. Support for arbitrary tape accessing functions is a current focus of the StreamIt compiler group. A scheduler can handle both external and helper functions with tape access; the overall work rate of a filter should reflect any tape access caused by functions it calls. This section gives examples showing this feature's importance.

External Functions with Tape Access Data compression formats such as JPEG and MPEG-2 pack data together as tightly as possible, ignoring word and byte boundaries in a data stream except for certain byte-aligned **escape codes** used to help a parser detect its position in a data stream. Even uncompressed formats, such as BMP [11] pack some configuration data. Parsers need functions that allow them to specify that a sequence of N bits appearing in the bitstream should be consumed and stored as some data type, such as an integer. The left half of Fig-

ure 8-1 shows example code from an MPEG-2 parser with external functions. The alternative approach without external functions appears in the right half[1].

Helper Functions with Tape Access Helper functions with tape access allow for cleaner code. They differ primarily from external functions with tape access in that they would be private to the filter that contains them and could manipulate internal state within that filter. The work function from the filter responsible for motion prediction is 103 lines and is difficult to read or edit (the code is omitted due to length). With tape-accessing helper functions it would decompose nicely into a work function and three helper functions, each of 20 to 30 lines.

8.3 Messaging Interfaces

In the current version of StreamIt, a filter `X` declares the types of messages it receives, and an associated `portal<X>` is implicitly declared. Only filters of the same type may subscribe to the same portal, so redundant portals must be declared if different parts of a computation need the same sets of messages. The StreamIt group has recognized this limitation and the upcoming language revision will introduce messaging interfaces, conceptually similar to Java's interface declaration, that decouple portal specifications from subscriber implementations. An interface declares a set of valid messages, and any filter which implements the interface may subscribe to portals of the interface's type.

Separating portal and subscriber definitions removes the dependency that senders broadcasting messages to a portal of type `portal<X>` have on subscribers that implement `X`. This dependency limits the expression of large applications and I provide empirical evidence that motivates implementation efforts for this feature. Consider again Tables 6-12 and 6-13. The number of distinct message subscriber types demonstrates the need for interfaces. Without interfaces, one must write each mes-

[1] The particular ability to pop, push, and peek multiple bits at a time from (or to) bitstreams, as illustrated in this example, is needed so frequently that all of the implementations described in this paper assume that these global functions exist and must be preprocessed before being sent to the StreamIt compiler.

sage sender after implementing all of its message receivers. In the encoder, enough components exchange information about picture and block metadata that message passing without interfaces became unmanageable and many parameters were embedded in the data stream. Interfaces enable one to write an MPEG-2 bitstream parser before writing — or even determining — the many downstream components that will need to receive control parameters from the parser.

8.4 Programmable Splitjoins

The current language semantics support roundrobin and duplicate splitters, and roundrobin joiners. Roundrobin components need not send or receive identical amounts on each of their substreams, but the rates must be statically determined. Some applications require computations which dynamically switch between several modes of computation based on data. Messaging can address the issue when the modes of computation are very similar. However, in cases where the computations differ significantly, it would be more natural to place independent filters in parallel and have a splitter select one of the filters for each set of data to be processed. This provides a need for programmable splitters and joiners which can receive messages that determine the data streams to which they should output. I first give an example where some kind of **switch** splitter that sends to one of its channels would suffice, and then show a case where programmable splitters and joiners are needed.

Switch Splitjoins Figure 8-2 shows the inverse quantization subgraph as it is realized in the MPEG-2 decoder. The inverse quantization process uses two different transforms depending on whether a block is intra coded or residually coded. The current implementation uses a splitjoin to duplicate each quantized block and process it for both block types. The results are interleaved and a downstream filter takes in both results and outputs one of them. Messages to the decision making filter from the upstream parser control which result gets output.

This approach results in needless computation since only one of the two inter-

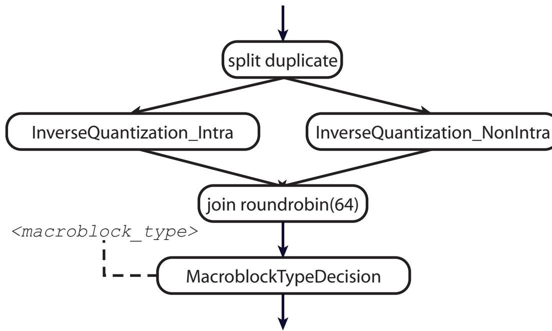


Figure 8-2: Inverse quantization subgraph with a duplicate splitter and roundrobin joiner.

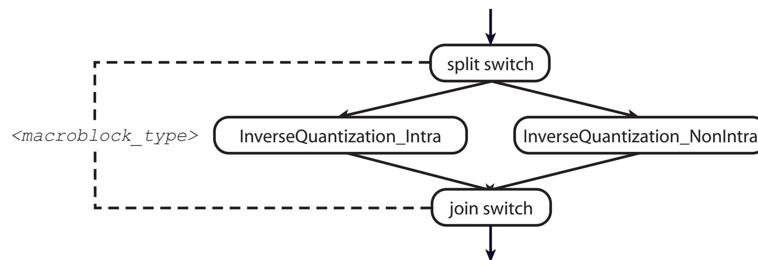


Figure 8-3: Inverse quantization subgraph with switch splitters and joiners.

mediate filters needs to execute. Dataflow is also complicated by the interleaving and filtering of data. In this case, a paired switch splitter and joiner are desirable. The idea is that a switch splitter and joiner with coordinated behavior would receive messages which cause them to change which channel processes the data. Figure 8-3 shows how the subgraph would appear with a switch splitter and joiner.

User Programmable Splitters Motion estimation in an MPEG-2 encoder, previously described in Section 5.3, illustrates the need for a programmable splitjoin. As implemented in Figure 5-3, a duplicate splitter sends data to each of the compression filters. A roundrobin joiner interleaves each of the compression results and sends them to a downstream filter which determines the best encoding technique and emits only one of the compression results. However, this results in extra work. For I pictures one need not try motion estimation and for P pictures one need not try backward motion prediction. With only duplicate splitters and roundrobin joiners

one is forced to send and receive data from the unneeded compression filters and then discard the results.

In an ideal implementation, the splitter and joiner would be programmable and messages about picture type would dictate which, and how many, internal streams processed the data. Figure 8-4 illustrates a stream graph with this form. A switch splitter will not suffice because multiple streams within the splitjoin must receive the input in certain cases. The joiner is also the ideal place to make the decision about which candidate blocks are used. For any given picture type, it must pop off the candidates from the valid input lines and output the one that exhibits the best compression. Note that the programmable splitter and joiners for this subgraph will be dynamic rates. However, the motion estimation subgraph is still static rate with respect to the rest of the stream graph and has a well-defined internal schedule. I explain how this scheduling information can be exposed to the compiler in the next section.

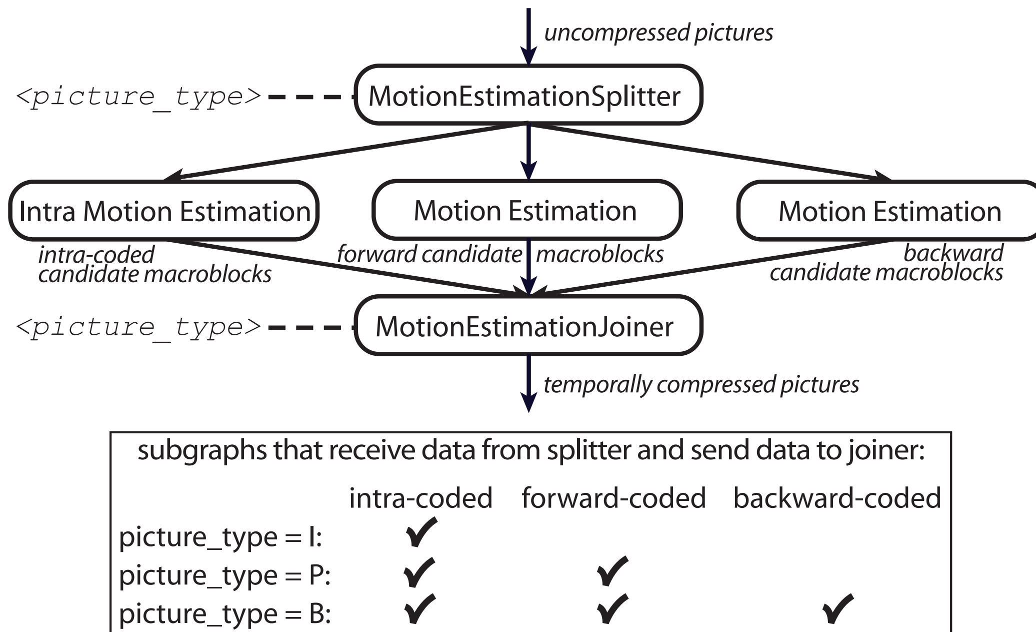


Figure 8-4: Motion estimation stream subgraph with a programmable splitter and joiner.

Another part of the encoder that would benefit from programmable splitjoins is the `ReferenceFrameHandler` responsible for decoding the recently encoded reference

pictures and sending them upstream to the motion estimation filters. The subgraph for this component appears in Figure 8-5.

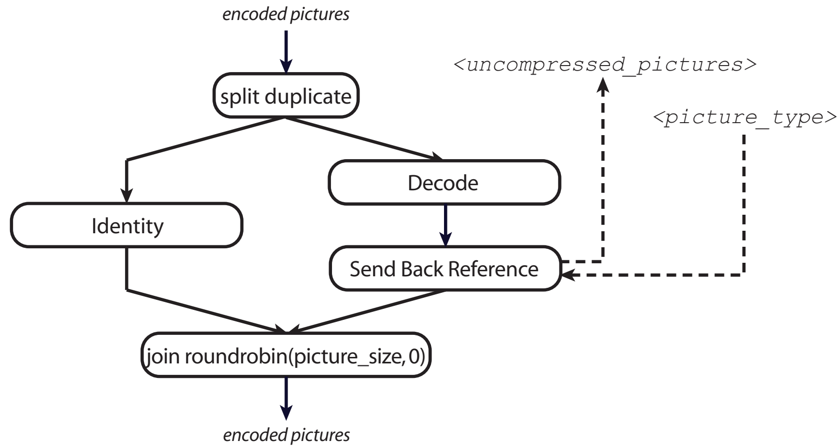


Figure 8-5: Subgraph for decoding reference pictures and sending them upstream.

As currently realized, both reference pictures and non-reference pictures go to a duplicate splitter. One duplicate of the encoded picture passes back to the joiner unchanged, and the other goes to a subgraph that first decodes the picture and then sends it as an upstream message to the motion estimation filters. The subgraph for decoding the picture looks similar to the MPEG-2 decoder pipeline.

One problem with this behavior is that the filter responsible for messaging back the reference frame must determine whether or not to send the picture or merely discard it, based on the picture type. This is problematic because many pictures are discarded, yet they are still properly decoded. A programmable splitter could receive picture type messages and selectively send only the reference frames to the decoding subgraph. This would avoid unnecessary computation.

8.5 Named Work Functions

Many filters whose work function behavior is dependent on stream parameters have a similar implementation pattern: state variables are declared and are used as parameters or control flow indicators in the work function, and only ever modified in message updates.

A more intuitive mechanism for the programmer would be to allow multiple named work functions or a single named work function that declares parameters (like a function or filter declaration). These named work functions would act as message receivers, and other portals could directly call the work functions through the teleport messaging mechanism.

From a sending filter's standpoint, messaging is unaffected. Whether a receiver implements a message handler as a named work function or a message handler is irrelevant to the messaging semantics. When a filter with a named work function receives a message, it would change the filter's mode of execution to the named work function, using the work function parameters if they exist.

Using named work functions simplifies the programming of the quantization, motion estimation, and picture reordering filters in the encoder, and the inverse quantization and motion prediction filters in the decoder. This feature also makes filter behavior more transparent to the compiler by exposing scheduling information about otherwise dynamic rate components. For instance, some dynamic rate filters could be declared as static rate filters if the different named work functions had different pop and push rates.

As an example, I consider the MPEG-2 encoder's motion estimation subgraph realized using programmable splitters and joiners, previously described in Section 8.4. As Figure 5-3 points out, the splitter and joiner will need to have different IO rates depending on the picture type, because different numbers of the splitjoins internal streams may need to receive picture data. With dynamic splitter and joiner IO rates, a compiler would be unable to intelligently schedule the subgraph's execution.

Suppose the splitter and joiner could each be implemented using three named work functions. Each work function would have a different but static rate declaration. By sharing the same interface and subscribing to the same message portal, this would expose the scheduling information to the compiler, which could realize that the splitjoin viewed as a whole has a static IO rate with respect to the rest of the graph.

8.6 Stream Graph Reinitialization

The biggest limitation of StreamIt 2.0 codec implementations is a requirement placed on video and image parameters that determine stream graph topology. These parameters must be declared at compile-time. Image size and chroma format must be hardcoded and changes to these values require that the source code be modified and the program recompiled. This limitation can be removed by allowing portions of a stream graph to reinitialize at runtime. Others have had success allowing actors to reinitialize at runtime and change their internal state [46], and it should be possible to allow topology changes at runtime as well. I believe this language feature is necessary and the StreamIt group is introducing the ability in StreamIt 3.0.

8.7 Stream Graph Draining

The streaming model of computation assumes that input and output streams are infinite and actors will process continuously with a steady state behavior. However, real world applications such as MPEG-2 have definite stream endings, and the end of life behavior of a stream graph is poorly determined. The MPEG-2 decoder, for instance, needs to be *drained* so that filters internally buffering data release their buffers. An example of this is the filter responsible for picture reordering (previously shown in Figure 4-9) which buffers one reference frame internally. For this frame to get released, the current implementation requires that the first filter in the pipeline, after detecting its final input, send dummy data items through the pipeline to empty out the remaining buffers. However, this behavior is non-ideal because spatial and temporal decoding are performed on data whose only purpose is to cause the downstream filter to release buffered data items. Just as the `prework` keyword helps a stream-graph reach a steady-state behavior at application startup, a similar mechanism (e.g., `postwork`) is needed at the end of an application's lifetime.

Chapter 9

Conclusions

Compression schemes play a key role in the proliferation of multimedia applications and the digital media explosion. At the same time, applications must be written for a plethora of unique parallel architectures. This thesis has shown that stream programming is an ideal model of computation for realizing image and video compression schemes. For this domain, stream programming boosts programmer productivity and enables scalable parallel execution of an application on a variety of architecture targets. Streaming language features allow a programmer to efficiently express a computation and expose parallelism, enabling a compiler to provide scalable performance.

This thesis shows these points through the following contributions: (*i*) clean, malleable, and portable MPEG-2 encoder and decoder implementations expressed in a streaming fashion, (*ii*) an analysis showing that a streaming language improves programmer productivity, (*iii*) an analysis showing that a streaming language enables parallel execution, (*iv*) an enumeration of the language features that are needed to cleanly express compression algorithms, (*v*) an enumeration of the language features that support large scale application development and promote software engineering principles such as portability and reusability.

This work was performed in the context of the StreamIt programming language, for its ability to express streaming computations, and the MPEG-2 video compression scheme. However the work is relevant to the domain of multimedia codecs,

including JPEG and MPEG-4. Currently the H.264 compression scheme is poised to supersede MPEG-2 video compression; assuming the language suggestions in this paper make their way into streaming languages, the most interesting research direction would be scalable, portable, and malleable implementations of H.264 codecs, expressed in a streaming fashion.

9.1 Future Research

Realtime processing is an important part of embedded systems and video codecs. StreamIt specifies timing constraints through message delivery semantics but these are logical scheduling constraints divorced from any absolute notion of time. No constraints can be specified regarding the maximum latency from stream graph input to output. One direction for future work would be to examine these realtime performance issues.

MPEG-2 is interesting from a language and compiler research perspective, but hardware advances have made MPEG-2 coding relatively easy. However, the new H.264 standard presents serious challenges to modern architectures and is designed to push performance well into the future. H.264 is the video coding scheme specified in Part 10 of the MPEG-4 video specification [29]. H.264 reflects many advances in video and image compression research. The major specification changes and their consequences for a stream-based implementation are detailed.

Video Organization

- Macroblocks can occur in any order within a picture. This will require additional messages associated with each macroblock indicating its position within an image.
- Macroblocks may be composed of blocks of many different sizes (8x8, 16x8, 16x16, 8x16), and block sizes may change from macroblock to macroblock. This will make programmable splitjoins and dynamic filter reconfiguration especially important as language features.

- Blocks are always composed of 4x4 subblocks, which have their own motion vectors.

- I and P pictures are no longer automatically considered reference pictures; instead, pictures are explicitly declared as reference pictures and may be used as reference pictures for an indeterminate length of time. The picture reordering filter will be substantially more complicated.

Spatial Coding

- The DCT is replaced by a wavelet transform. The wavelet transform incorporates the quantization scale factors. The transform is simple to implement and simplifies the spatial decoding process from a programmer's standpoint, although little can be reused from MPEG-2.

- A block-aliasing mechanism removes edge-artifacts between macroblock boundaries. The spatial coding pipeline can be naturally extended to include an additional filter or subgraph that handles this transformation.

Motion Prediction

- Motion vectors may now refer to data contained in other macroblocks in the same frame. Instead of referring to the previous 2 decoded reference frames, P and B pictures may now refer to any of the past 32 decoded reference frames. The optimization to share read-only message sent data between filters will be necessary.

- Because blocks have variable size and are composed of subblocks with their own motion vectors, the motion prediction filters are substantially complicated in both encoders and decoders. An encoder will only benefit from the huge variety of prediction options if it can expose a high degree of coarse grained parallelism in the motion estimation subgraph.

- Motion vectors include a scaling factor (for zooms) and an amplitude factor (for image fades). Motion vectors are also accurate to a quarter pixel resolution instead of half pixel resolution. This particular aspect of motion prediction can be accommodated by small changes to the existing prediction formation process.

Variable Length Coding

- The bit-stream syntax uses many new variable bit rate encoding algorithms which have higher compression rates than Huffman coding. The bitstream also makes a distinction about the importance of data and provides options for variable rate error correction. Bitstream parsing is therefore dramatically more complicated and good inter-language interfaces for this non-streaming computation are critical.

StreamIt should be well suited for MPEG-4 decoding and encoding. Encoding, in particular, should benefit from the extreme kinds of parallelization that StreamIt can expose. MPEG-4 provides a huge variety of motion prediction options for encoding any particular video and exploring the space of potential compression options will require a scalable parallel encoder implementation.

Bibliography

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

[2] Ishfaq Ahmad, Shahriar M. Akramullah, Ming L. Liou, and Muhammad Kafil. A scalable off-line MPEG-2 video encoding scheme using a multiprocessor system. *Parallel Computing*, 27(6), 2001.

[3] Ishfaq Ahmad, Yong He, and Ming L. Liou. Video compression with parallel processing. *Parallel Computing*, 28, 2002.

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

[5] Shahriar M. Akramullah, Ishfaq Ahmad, and Ming L. Liou. A Data-Parallel Approach for Real-Time MPEG-2 Video Encoding. *Journal of Parallel and Distributed Computing*, 30(2), 1995.

[6] Shahriar M. Akramullah, Ishfaq Ahmad, and Ming L. Liou. Performance of Software-Based MPEG-2 Video Encoder on Parallel and Distributed Systems. *IEEE Transactions on Circuits and Systems for Video Technology*, 7(4), 1997.

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

[8] I. Assayad and S. Yovine. Parallel model analysis and implementation for MPEG-4 encoder. In *Proc. of Embedded Processors for Multimedia and Communications II*, 2005.

[9] Ismail Assayad, Philippe Gerner, Sergio Yovine, and Valerie Bertin. Modelling, Analysis and Parallel Implementation of an On-line Video Encoder. In *1st Int. Conf. on Distributed Frameworks for Multimedia Applications*, 2005.

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

[11] BMP File Format. http://www.digicamsoft.com/bmp/bmp.html.

[12] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. In *SIGGRAPH*, 2004.

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

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

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

[16] Inmos Corporation. *Occam 2 Reference Manual*. Prentice Hall, 1988.

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

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

[19] Matthew Drake, Henry Hoffman, Rodric Rabbah, and Saman Amarasinghe. MPEG-2 Decoding in a Stream Programming Language. In *International Parallel and Distributed Processing Symposium*, Rhodes Island, Greece, April 2006.

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

[21] 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.

[22] 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.

[23] Thierry Gautier, Paul Le Guernic, and Loic Besnard. Signal: A declarative language for synchronous programming of real-time systems. *Springer Verlag LNCS*, 274, 1987.

[24] Michael Gordon and Saman Amarasinghe. Space-Time Multiplexing of Stream Programs. In *ASPLOS*, 2006.

[25] Michael Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Christopher Leger, Andrew A. Lamb, Jeremy Wong, Henry Hoffman, David Z. Maze, and Saman Amarasinghe. A Stream Compiler for Communication-Exposed Architectures. In *ASPLOS*, 2002.

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

[27] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proc. of the IRE*, 40(9), September 1952.

[28] ISO/IEC 13818: 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.

[29] ISO/IEC 14496-10: Information technology — Coding of audio-visual objects: Advanced video coding. International Organization for Standardization, 2002.

[30] ITU T.81: Information Technology — Digital compression and coding of continuous-tone still images — Requirements and guidelines. International Telecommunication Union, 1992.

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

[32] Tom Jacobs, Vassilios Chouliaras, and David Mulvaney. Investigation of Thread-Level Parallelism in the Architectural Complexity Reduction of MPEG2, XviD and H.264 Video Encoders. In *2nd ESC Division Mini Conference*, 2005.

[33] Ujval J. Kapasi, Scott Rixner, William J. Dally, Brucek Khailany, Jung Ho Ahn, Peter Mattson, and John D. Owens. Programmable Stream Processors. *IEEE Computer*, 2003.

[34] Michal Karczmarek. Constrained and Phased Scheduling of Synchronous Data Flow Graphs for StreamIt Language. S.m. thesis, Massachusetts Institute of Technology, Cambridge, MA, December 2002.

[35] Dong-Ik Ko and Shuvra S. Bhattacharyya. Dynamic Configuration of Dataflow Graph Topology for DSP System Design. In *ICASSP*, 2005.

[36] Dong-Ik Ko and Shuvra S. Bhattacharyya. Modeling of Block-Based DSP Systems. *Journal of VLSI Signal Processing*, 40(3), 2005.

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

[38] Andrew A. Lamb, William Thies, and Saman Amarasinghe. Linear Analysis and Optimization of Stream Programs. In *PLDI*, 2003.

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

[40] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.

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

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

[43] Man-Lap Li, Ruchira Sasanka, Sarita V. Adve, Yen-Kuang Chen, and Eric Debes. The ALPBench Benchmark Suite for Complex Multimedia Applications. In *IEEE Int. Symp. on Workload Characterization*, 2005.

[44] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: A System for Programming Graphics Hardware in a C-like Language. In *SIGGRAPH*, 2003.

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

[46] Stephen Neuendorffer and Edward Lee. Hierarchical Reconfiguration of Dataflow Models. In *Conference on Formal Methods and Models for Codesign*, 2004.

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

[48] Chanik Park, Jaewoong Jung, and Soonhoi Ha. Extended Synchronous Dataflow for Efficient DSP System Prototyping. *Design Automation for Embedded Systems*, 6(3), 2002.

[49] Nuria Pazos, Paolo Ienne, Yusuf Leblebici, and Alexander 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.

[50] Fernando L. Pelayo, Fernando Cuartero, Valentín Valero, Diego Cazorla, and Teresa Olivares. Specification and Performance of the MPEG-2 Video Encoder by Using the Stochastic Process Algebra: ROSA. In *17th UK Performance Evaluation Workshop*, 2001.

[51] Claus Schneider. Executable Specification for Multimedia Supporting Refinement and Architecture Exploration. In *25th Euromicro Conference*, 1999.

[52] Janis Sermulins, William Thies, Rodric Rabbah, and Saman Amarasinghe. Cache Aware Optimization of Stream Programs. In *LCTES*, 2005.

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

[54] SLOCCount. http://www.dwheeler.com/sloccount/.

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

[56] StreamIt Language Specification, Version 2.0. http://cag.lcs.mit.edu/streamit/papers/streamit-lang-spec.pdf.

[57] StreamIt MPEG-2 Documentation. http://cag.csail.mit.edu/streamit/mpeg/.

[58] MPEG-2 Encoder and Decoder (StreamIt Code). http://cag.csail.mit.edu/streamit/mpeg/mpeg_streamit_codec.tar.gz.

[59] Jim Taylor. Standards: DVD-video: multimedia for the masses. *IEEE MultiMedia*, 6(3), July–September 1999.

[60] Michael Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Benjamin Greenwald, Henry Hoffman, Jae-Wook Lee, Paul Johnson, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matthew Frank, Saman Amarasinghe, and Anant Agarwal. The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs. *IEEE Micro*, May 2002.

[61] Michael Taylor, Walter Lee, Jason Miller, David Wentzlaff, Ian Bratt, Benjamin Greenwald, Henry Hoffman, Paul Johnson, Jason Kim, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matthew Frank, Saman Amarasinghe, and Anant Agarwal. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. In *International Symposium on Computer Architecture*, Munich, Germany, June 2004.

[62] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A Language for Streaming Applications. In *Int. Conf. on Compiler Construction*, 2002.

[63] William Thies, Michal Karczmarek, Janis Sermulins, Rodric Rabbah, and Saman Amarasinghe. Teleport messaging for distributed stream programs. In *PPoPP*, 2005.

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

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

[66] Bhaskaran Vasudev and Konstantinides Konstantinos. *Image and Video Compression Standards*. Kluwer, 1997.

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