Implementation Techniques for Continuous Media Systems and Applications

by

Brian Christopher Smith

A.B. (University of California at Berkeley)1986
M.S. (University of California at Berkeley)1990

A Dissertation Submitted in Partial Satisfaction of the

Requirements for the Degree of

Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY of CALIFORNIA at Berkeley

Committee in charge:

Professor Lawrence A. Rowe (chair)

Professor Carlo H. Sequin

Professor Alice Agogino

1994

Implementation Techniques for Continuous Media Systems and Applications

by

Brian Christopher Smith

Abstract

Implementation Techniques for Continuous Media Systems and Applications

by

Brian Christopher Smith

Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor Lawrence A. Rowe, Chair

In this thesis, I investigate issues in the development of continuous media (CM) applications. CM applications process, transport, or store CM data such as digital audio and video.

Introduction of video into applications will require support for sophisticated video effects such as image processing, composition and digital filtering. Traditional image processing and composition algorithms operate on uncompressed images, while video is typically stored and delivered in a compressed form. High computational cost, along with the complexity of video decompression, makes traditional algorithms too slow for interactive use on video data.

This thesis describes and evaluates a new family of algorithms for computing video effects that run one to two orders of magnitude faster than their traditional counterparts. The performance increase is achieved by performing the operations directly on the compressed data. Using these algorithms, I show that simple special effects can be computed in real time on current generation workstations.

In addition to developing algorithms for special effects, I describe and evaluate a toolkit for constructing distributed CM applications, called CMT, and a best effort network protocol for delivering CM data, called cyclic-UDP.

CMT is an extension to the Tcl/Tk graphical user interface toolkit. It simplifies the creation and coordination of the multiple processes in a distributed CM playback system that supports delivery and synchronized playback of multiple CM streams from one or more file servers. Besides making playback applications easier to construct, the toolkit also provides a foundation on which new delivery, display, and synchronization protocols can be built and tested.

CMT uses a best-effort network protocol called cyclic-UDP for delivering CM data to applications. Cyclic-UDP differs from other CM transport protocols in two ways. First, it works well in local, metropolitan, and wide area network environments, adapting itself dynamically to each new scenario. Second, it uses properties of the data being transported to reduce to effect of network congestion, both chronic and acute, to lowered fidelity at the display. For example, network congestion will cause the frame rate of a video stream to drop or decrease the effective sampling rate of an audio stream. In other protocols, congestion causes bursty frame loss where the video streams "stops" until congestion abates. Cyclic-UDP achieves improved quality by supporting frame priority which gives high priority frames a better chance of delivery than low priority frames. When used in combination with protocols specifically tuned to a particular media format, called *media-specific* protocols, cyclic-UDP can amortize the degradation in output quality caused by burst packet losses over a long period. In this thesis, I describe and experimentally evaluate three media-specific protocols, one for MPEG video, one for motion-JPEG video, and one for uncompressed audio.

Finally, using CMT and cyclic-UDP, I show that it is possible to play good quality video using existing network and operating system technology by building a working playback application, called the CM Player.

To my parents

Table of Contents

List of Figures

List of Tables

Acknowledgments

Many people go into the production of a Ph.D. thesis. Adequately acknowledging these people is inherently futile, since in many cases their contribution far exceeds my capacity for expression.

My peers, Joseph Konstan, Steve Seitz, Steve Yen, Beverly Sobleman, Robert Wahbe, Steve Lucco, John Boreczky, Steve Smoot, and Ketan Patel have patiently listened to my unending pontification, have given insight, and have provided encouragement and companionship.

My fellow martial artists and teachers, Kevin Stefanik, Dr. Min, Jimmy Kim, Elbert Kim, Chuck Buhs, Victoria Wohl, Jason DD, and countless others have helped tone, focus, and discipline my mind and my spirit. They have been faithful companions through challenging times.

My fellow adventurers, Mike, Rob, Andy, and the UC Hang Gliding Club, have balanced my life and have taught me to control my fear. They have given me a multitude of hair-raising tales and memories that supported me while chained to my desk.

My family, Carl, Barbara (Mom and Dad), Steve, Mark, Bruce, Mary, Noreen, Jenelle, Michelle, Kristina, and Kevin, has always been there, supporting me, financially and emotionally, through all these years and trials. Despite my lack of contact at times, they were always there, and I knew it.

My research advisor, Larry Rowe, has spent innumerable hours discussing ideas and refining my research skills. He has taught me what I think is the more intangible idea in any scientist's career, the recognition of "interesting research." He has helped me to negotiate the maze of obstacles that is graduate school, he has refined my research and presentation skills, and above all things, he is my friend.

Finally, my “significant other,” Tracy, has changed me in ways not yet realized. She has altered my view of the world and has taught me new ways of thinking. I would not have made it though the last, most difficult years without her love and confidence.

Chapter 1
Introduction

Technological developments of the last decade in networking, CPUs, and compression standards, have enabled the creation of *continuous media* (CM) applications. CM applications support the retrieval, manipulation and display of CM data, which is data that changes continuously over time such as digital audio, digital video and animation. Many CM applications have been cited in the literature [3,48,73], including:

1. Video teleconferencing systems that allow two or more users to hold a conversation with each other.
2. Computer supported cooperative work (CSCW) systems that allow two or more users to collaborate on the editing of a document or group of documents.
3. Distributed lecture systems that broadcast lectures to an audience on a computer network.
4. Hypermedia systems that allow users to browse multimedia information in a non-linear format.
5. Video on-demand services that provide movie delivery over networks to the home.
6. Educational courseware that allow students on-line search and retrieval of digitized lectures, simulations, homeworks, lecture notes, and other course-related materials.
7. Audio/video editing that allow users to create CM documents.

This thesis discusses two issues in the development of CM applications: CM toolkits and video special effects.

Despite widespread interest, only a few working CM applications have been built and studied. Research has been hindered by the difficulties encountered in creating the applications. Although they share many common functions, the diversity of user interfaces and underlying mechanisms, coupled with the real-time constraints of CM applications and their propensity to push the performance of the current technology to its limits, has made real CM applications difficult and costly to produce. The situation is similar to that felt by the graphical user interface (GUI) community in the mid 1980s when programmers wanted to write a variety of programs with GUIs, but found it required tens or hundred of thousands of lines of code.

The solution to the problem in the GUI community came with the advent of GUI toolkits. These toolkits provided a library of commonly used functions and a framework for composing the functions. With the use of GUI toolkits, the number of lines of code required for the user interface (UI) portion of an application was reduced typically by several orders of magnitude.

I used the same toolkit strategy to address the software engineering problem posed by CM applications by extending an existing GUI toolkit, the Tcl/Tk toolkit [55] to support abstractions for CM applications. The changes I made include modifying the event processing mechanism to support soft real-time scheduling and adding a distributed programming infrastructure, called Tcl-DP, that simplifies the creation of distributed applications. Using the extended toolkit as infrastructure, I built a framework called the CM Toolkit (CMT) to allow a variety of distributed CM playback applications to be built.

In addition to simplifying the creation of CM playback applications, CMT provides a clean, modular structure that allows other researchers to experiment with transport and scheduling mechanisms. Other researcher are currently using CMT as a platform to test their ideas (e.g., a desktop video conferencing system [14] and the Priority Encoded Transmission project [1]).

Aside from the software engineering difficulties, the data volume and bandwidth required for CM applications raises a host of problems. Broadcast quality video data uses a bandwidth of 27 MBytes/sec, so storage of a one hour program requires over 92 GBytes of storage. The limitations of current networks and storage devices require that the data be compressed which raises the issue of compression standards. Fortunately, several compressions standards have emerged to answer this need [57, 82, 44].

Although these standards are an essential part of a solution to the storage and delivery problems of CM systems, most compression schemes are complex and computationally intensive. Therefore, decompression must be done either by dedicated hardware or by carefully optimized software.

Even with compression hardware assistance, another, more fundamental problem, arises. Users want to edit CM data just like they edit text and graphics. Soon, they will want to perform special effects, image enhancement, and other traditional image processing functions. The overhead of decompression, coupled with the high throughput of the decompressed data, makes image processing too time-consuming for interactive use on current workstations without special purpose hardware to accelerate these functions.

For example, suppose you implemented some simple image processing algorithms on a video sequence with the goal of making them operate in real-time. Since the algorithm must read 27 Mbytes/sec, and write the same amount, the memory bandwidth is about 54 MBytes/sec. Assuming the CPU performs a load, save, and arithmetic instruction on each byte in the input stream, the CPU speed would have to be at least 81 MIPS. This calculation ignores overhead associated with address computation, decompression, and the fact that the image processing operation is likely to be much more complex than a single arithmetic instruction. With these added complexities, real-time image processing on video data becomes impossible on today's machines.

This problem can be solved if the image processing operation can be performed directly on the compressed video data. Since the volume of data is typically one to two orders of magnitude less than the uncompressed image size, and the computationally complex decompression is avoided until the image is displayed, image processing becomes feasible, provided a simple operation on the compressed data can be found that matches the image processing operation on the decompressed data.

I have developed an algebra that allows a variety of useful image processing operations to be mapped to compressed data operations. The algebra is applicable both to local operations and global operations. A local operation is one where the value of a pixel in the output image is determined by the value of a single corresponding pixel in the input image(s), and a global operation is one where the value of a pixel in the output image is determined by the value of two or more pixels in the input image. Examples of local operations are contrast enhancement, dissolves, video composition (as in TV-news weather forecasts), and the like. Examples of global operations are geometric operations such as scaling, rotation, translation, and image processing operations such as edge detection and image enhancement.

Experimentation with this technique shows that operations implemented on the compressed representation are one to two orders of magnitude faster than methods that decompress the image, compute the operation, and compress the result. The compressed domain operations are suitable for interactive use on next generation workstations and for home computers (e.g., set-top boxes and video game machines).

The remainder of this dissertation is organized as follows. In chapters 2 and 3, I describe the algebra for processing compressed images outlined above. In these chapters, I also evaluate the technique's performance on current generation machines and discuss the data structures used in an efficient implementation. Chapter 4 describes the architecture, data model, and implementation of CMT, and chapter 5 describes experiments with the network transport protocol used in CMT.

Chapter 2
Local Image Operations

In digital video effects (DVE) processing, one or more *input* images are combined to create an output image. These images are typically the individual frames of one or more video streams. This type of processing, used in robot vision [35], digital signal processing [45], computer graphics [61,23], and video editing systems, consumes large amounts of computing resources.

For example, consider the resources needed to process a 640 pixel by 480 pixel, 24 bit video sequence at 30 frames per second (fps). Each pixel in the input image must be read, used, and stored. Each image requires 900 KBytes of storage, so this process involves reading 27 MBytes/sec and writing an equal amount, for an aggregate bandwidth of 54 Mbytes/sec. Since the unit of data is one byte in this example, each operation on a conventional RISC architecture will require one instruction to load the byte, at least one to modify the byte, and one to store the result, yielding a lower bound on the CPU requirements of 81 MIPS. With the additional overhead of loop counters, address arithmetic, and operations that require several instructions per pixel, it is clear that several hundred MIPS is required for all but the most trivial processing.

The Cost of Decompression

To complicate matters even further, video sequences are almost always stored in a compressed format such as MPEG [44] or motion JPEG [57,82]. To measure the cost of decompression and compression, I instrumented a public domain JPEG software decompression program [43] to count the number of instructions executed during both compression and decompression. I used the pixie [60] profiling software for the MIPS R3000 family of processors to instrument the code, and ran an experiment on a DECstation 5000/125 workstation. In the experiment, 621 different images were compressed and decompressed, and the instructions required for each operation were counted and classified into three groups: arithmetic operations (arithmetic, shift, and logic instructions), memory operations (load and store instructions), and miscellaneous instructions (including control flow). Figures 2-1 and 2-2 show the number of operations in each class required for compression and decompression. Analysis of this data shows that decompression requires an average of 283 ($\sigma = 17$) instructions for each pixel in the input image, and compression requires 314 ($\sigma = 10$) instructions for each pixel in the output image. This means that to decompress the 640 by 480 pixel video signal for DVE processing, $640 \times 480 \times 30 \times 283 \cong 2600$ MIPS of RISC processing is required to decompress the image. If the DVE is to be stored, or if it is computed in an intermediate node in a computer network and then transmitted, the result of the DVE must also be compressed, requiring an additional $640 \times 480 \times 30 \times 314 \cong 2800$ MIPS. Algorithms that fully decompress an image, apply a DVE, and compress the result are called *brute force* algorithms.

In this chapter and the next, I examine a software approach to computing DVEs that operates directly on the compressed video data. I will show that this approach uses much fewer computing resources, so that near real-time DVEs can be performed in software.

This chapter examines a class of DVEs called *local DVEs* (LDVEs). LDVEs are DVEs where the value of a pixel in the output image is determined by the value of the corresponding pixel(s) in the input image(s). The next chapter looks at more general DVEs. The rest of this chapter is organized as follows. In section 2.1, I review a typical transform-based compression technique for images, the JPEG compression standard. In section 2.2 I define the basic algebraic operations needed to implement LDVEs, and show how to implement these operations on compressed JPEG data, and in section 2.3 I report the results from an experi-

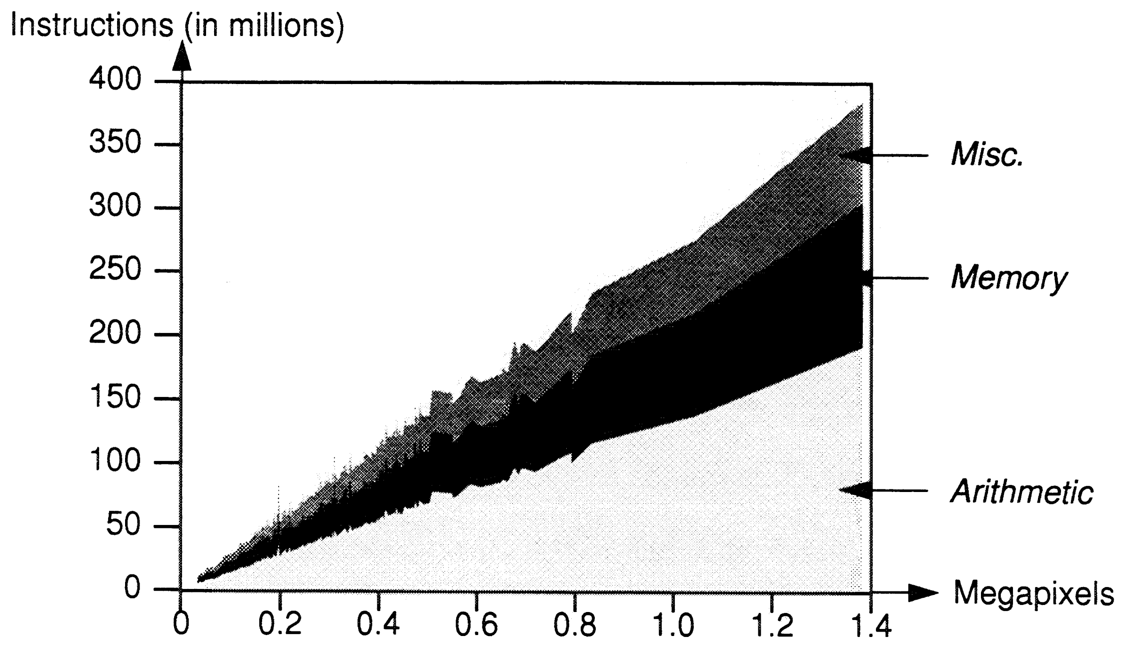


Figure 2-1: Number of instruction required for decompression

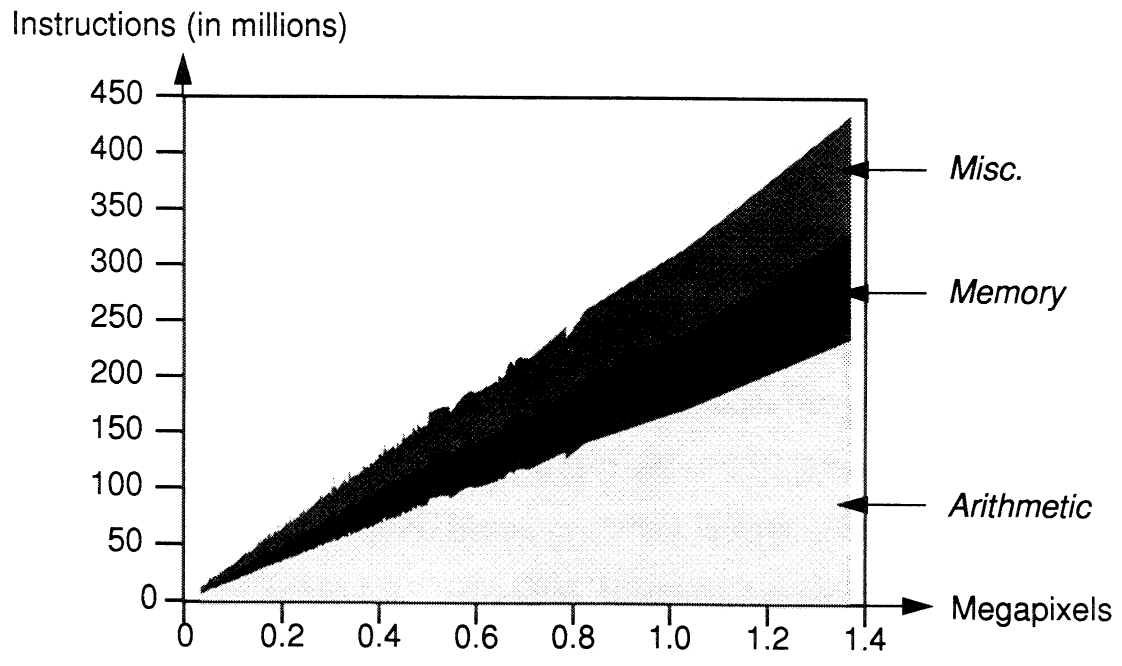


Figure 2-2: Number of instruction required for compression

mental implementation of these techniques.

2.1 JPEG Coding

This section describes the compression model used in transform based coding, beginning with a general review of transform coders and then continuing with a brief description of the CCITT Joint Photographic Expert Group (JPEG) proposed standard for transform based image coding. A detailed description of the JPEG algorithm is available elsewhere [57,82]. A detailed description of image formats is presented by Foley and Van Dam [23]. Other transform coding techniques are discussed in [45,37].

Transform Based Coding

A common technique for image compression is *transform based coding*[1]. In a typical transform coder, the pixels of the image are treated as a matrix of numbers. A linear transform, such as the discrete cosine transform (DCT [63]), is applied to this matrix to create a new matrix of coefficients. To recover the original image, the inverse linear transformation is applied.

The transformation has two effects. The first effect is to concentrate the energy of the image so that a large number of the transformed coefficients are nearly zero. The second effect is to spectrally decompose the image into high and low frequencies. Since the human visual system is less receptive to some frequencies than others, some coefficients can be more crudely approximated than others without significant image degradation.

A common way to exploit the latter property is to *quantize* the coefficients. A simple way to quantize coefficients is to truncate low order bits from an integer

[1] All results in this section can be found in [45], and shall be stated without proof.

(e.g., by right arithmetic shifting). A method that provides more control over data loss than arithmetic shifting is to divide the value by a constant, the *quantization value*, and round the result to the nearest integer. An approximation of the original value can be recovered by multiplying the result by the quantization value. Larger quantization values lead to cruder approximations, but fewer significant bits.

When the DCT transformed coefficients of typical images are quantized in this way, most coefficients are typically zero. For example, measurements indicate that about 90% of the coefficients in a typical image compressed using this technique can be set to zero without noticeable degradation.

The JPEG algorithm

One standard for transform coding of still images is the JPEG standard. The remainder of this section briefly describes relevant features of JPEG and introduces associated terminology.

Suppose the source image is a 24 bit image 640 pixels wide by 480 pixels high, and it is composed of three components: one *luminance (Y)* and two *chrominance (I* and *Q*). That is, a triplet of 8 bit values (*Y,I,Q*) is associated with each pixel in the source image. Since each component is treated similarly, I will describe the algorithm for only one component (e.g., the *Y* component).

The JPEG algorithm consists of six steps:

1) Normalization The *Y* component is broken up into contiguous squares 8 pixels wide by 8 pixels high called *blocks*. Each block is an 8 by 8 matrix of integers in the range 0...255. The first step of the algorithm, called the *normalization* step, brings all values into the range -128...127 by subtracting 128 from each element in the matrix (this step is skipped on the *I* and *Q* components since they are already in the range -128...127). Let the resulting matrix be *y[i,j],* where $i, j \in 0 \ldots 7$. Figure 2-3 illustrates the relationship of *y[i,j]* to the whole image.

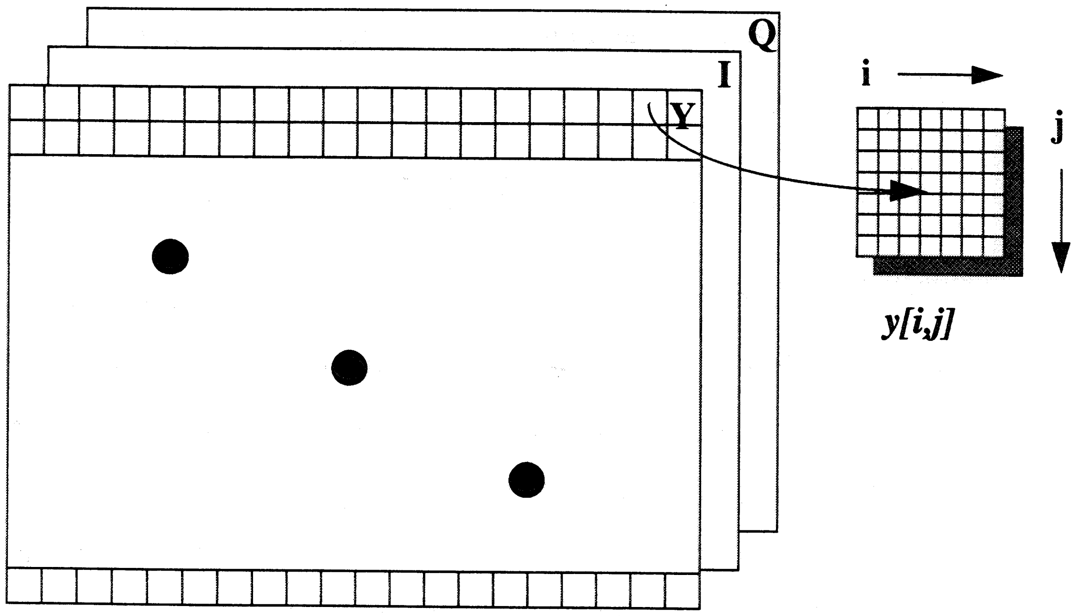


Figure 2-3: Definition of y[i,j].

2) DCT The second step in the algorithm applies the DCT to this 8 by 8 matrix, producing a new 8 by 8 matrix. This step is called the *DCT* step. The DCT is similar to the fast fourier transform in that the values in the resulting 8 by 8 matrix are related to frequency. That is, the lowest frequency components are in the upper left corner of the output matrix, and the highest frequency components are in the lower right. Let the new matrix be denoted $Y[u, v]$, with $u, v \in 0 \ldots 7$. By the definition of the DCT:

$$Y[u, v] = \frac{1}{4}\sum_{i}\sum_{j} C(i, u)\, C(j, v)\, y[i, j] \qquad \text{EQ 2-1}$$

where

$$C(i, u) = A(u) \cos\frac{(2i+1)u\pi}{16}$$

$$A(u) = \begin{cases} \frac{1}{\sqrt{2}} & \text{for } u = 0 \\ 1 & \text{for } u \neq 0 \end{cases}$$

3) Quantization The third step in the algorithm quantizes each element of $Y[u, v]$ by a value dependent on the frequency, called $q[u, v]$. This *quantization* step is defined by

$$Y_Q[u, v] = IntegerRound\left(\frac{Y[u, v]}{q[u, v]}\right) \qquad u, v \in 0\ldots7 \qquad \text{EQ 2-2}$$

The matrix of integer quantization values $q[u, v]$ is called the *quantization table* (QT). Different QTs are typically used for the luminance and chrominance components. The choice of the QT determines both the amount of compression and the quality of the decompressed image [82,26,58,83,40]. The JPEG standard includes recommended luminance and chrominance QTs which are the results of human factors studies. A common practice is to scale the values of these default QTs to obtain different image qualities. Specifically, given two images with QTs $q_1[u, v]$ and $q_2[u, v]$, then for all *u,v* and some constant gamma, it is often the case that

$$\frac{q_1[u, v]}{q_2[u, v]} = \gamma \qquad \text{EQ 2-3}$$

I will use this fact later.

4) Zig Zag Scan Step four of the algorithm converts the 8 by 8 matrix $Y_Q[u, v]$ from equation 2-2 into a 64 element vector $Y_{ZZ}[x]$ using the "zig zag" ordering shown in Figure 2-4. This ordering is a heuristic to cluster low frequency components near the beginning of the vector and high frequency components near the end. The vector Y_{ZZ} is called the *zig-zag vector,* and this step is called the *zigzag scan* step.

5) Run Length Encoding (RLE) In most images, the vector Y_{ZZ} will contain a large number of sequential zeros, so the next step in the algorithm, the *run length encoding* step, encodes the vector into (*skip, value*) pairs. *Skip* indicates how

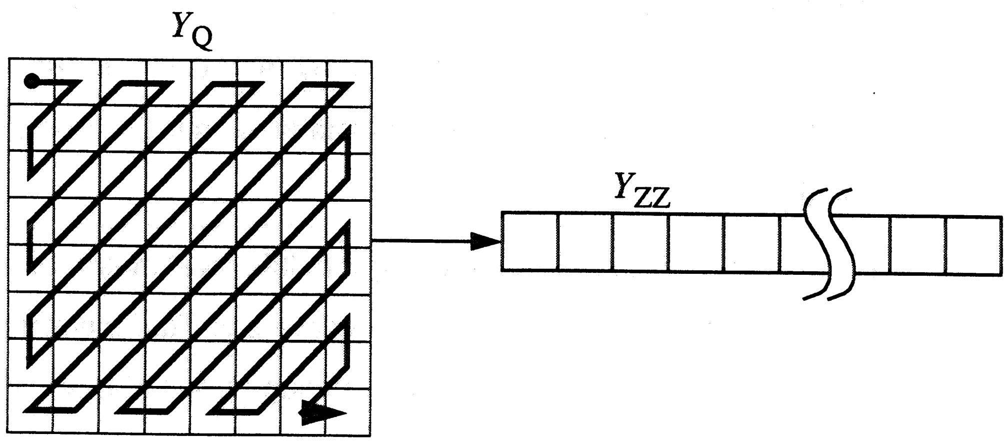


Figure 2-4: Zig Zag Scan Ordering

many indices in the Y_{ZZ} vector to skip to reach the next non-zero value, which is stored in *value*. By convention, the pair (0,0) indicates that the remaining values in Y_{ZZ} are all zero. This block is called a *run-length-encoded* (RLE) block, and each (*skip, value*) pair is called an *RLE value*. The RLE block is denoted $Y_{RLE}[x]$, with $Y_{RLE}[x]$.*skip* and $Y_{RLE}[x]$.*value* denoting the *skip* and *value* of the *x*th element in the array. The algorithms in this chapter operate on RLE blocks.

6) Entropy Coding In the final step, a conventional entropy coding method such as arithmetic compression or Huffman coding compresses the RLE blocks. Figure 2-5 graphically displays all the steps in the processing of one block.

The compression ratio is adjusted by altering the values in the QTs. Experi-

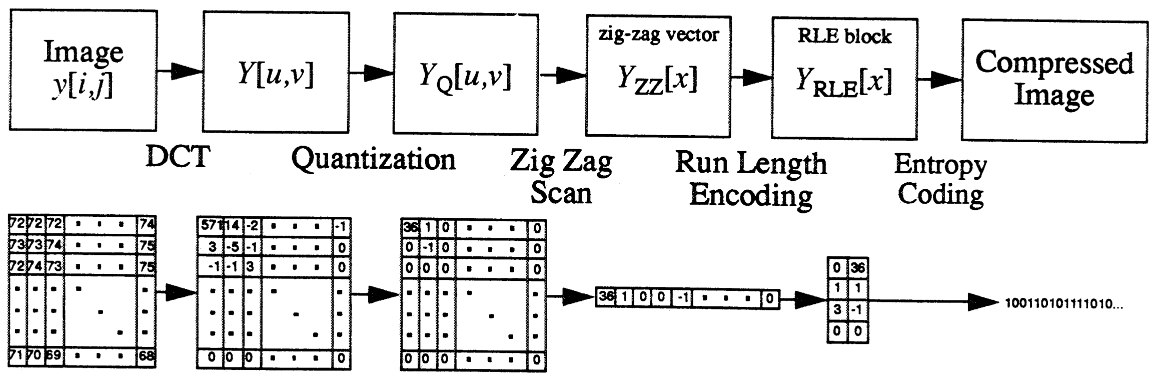


Figure 2-5: Compression of a Block

ence indicates that a compression ratio of about 24 to 1 (i.e., one bit/pixel) can be achieved without serious loss of image quality. At a compression ratio of 12 to 1 (i.e., 2 bits per pixel), the decompressed image is usually indistinguishable from the original [26].

Tracing figure 2-5 backwards (i.e., right to left) illustrates how to decompress the data. The first step of decompression recovers an RLE block from the entropy coded bit stream. By making a single pass through the RLE block $Y_{RLE}[x]$, the zig-zag vector $Y_{ZZ}[x]$ can be recovered. From $Y_{ZZ}[x]$, $Y_Q[u,v]$ can be reconstructed by inverting the zig-zag scan. Then each element of $Y_Q[u,v]$ is multiplied by $q[u,v]$ from the appropriate QT to recover an approximation of $Y[u,v]$. In the final step, the image block $y[i,j]$ is obtained from $Y[u,v]$ using the inverse DCT (IDCT):

$$y[i,j] = \frac{1}{4}\sum_u \sum_v C(i,u)\,C(j,v)\,Y[u,v] \qquad \text{EQ 2-4}$$

Note that equation 2-4 is very similar to equation 2-1, but the summation is over u and v rather than i and j.

2.2 Algebraic Operations

This section shows how the four algebraic operations of scalar addition, scalar multiplication, pixel-wise addition and pixel-wise multiplication of two images are performed on RLE blocks.

In the calculations that follow, we will be deriving equations of the form

$$H_{RLE} = \phi(F_{RLE}, G_{RLE}) \qquad \text{EQ 2-5}$$

where F_{RLE} and G_{RLE} are the RLE representations of the input images, H_{RLE} is the RLE representation of the output image, and ϕ is a real-valued function. In an implementation, the values stored in the H_{RLE} data structure would be integers, so the value returned by the function ϕ must be rounded to the nearest integer. To simplify the notation in the calculations that follow, this rounding will be implicit.

To further simplify the notation, all calculations will be performed on the quantized arrays $F_Q[u,v]$, $G_Q[u,v]$ and $H_Q[u,v]$. Since an RLE block is a data structure that represents these arrays, the derived equations will be valid on RLE blocks provided the appropriate index conversion is performed.

Other notational conventions are listed below:

1. Capital letters such as *F*, *G* and *H* indexed by *u, v* and *w* will represent compressed images.
2. Lower case letters such as *f, g,* and *h* indexed by *i, j* and *k* will represent uncompressed images.
3. Greek letters such as α and β will be used for scalars
4. QTs will be represented as arrays, with subscripts indicating the image. For example, $q_H[u,v]$ will stand for the QT of the compressed image *H.*
5. The letters *x*, *y*, and *z* will represent zig-zag ordered indices.

QTs will often be indexed by a single zig-zag index (such as *x*). In such cases the conversion to indices such as *[u,v]* is implied and will be clear from context.

Scalar Multiplication

Consider the operation of scalar multiplication of pixel values. In this operation, if the value of a pixel in the original image is $f[i,j]$, the value of the corresponding pixel in the output image $h[i,j]$ is given by

$$h[i,j] = \alpha f[i,j] \qquad \text{EQ 2-6}$$

Using the linearity of the JPEG compression algorithm and equation 2-6, it can be easily shown that the quantized coefficients of the output image, $H_Q[u,v]$, are just scaled copies of $F_Q[u,v]$. Specifically, it can be shown, using equations 2-1 through 2-4 and equation 2-6, that

$$H_Q[u,v] = \frac{\alpha q_F(u,v)}{q_H(u,v)} F_Q[u,v] \qquad \text{EQ 2-7}$$

where $q_F(u,v)$ and $q_H(u,v)$ are the QTs of the input and output images, respectively, and with the final integer rounding of the right hand side being implicit. In other words, to perform the operation of scalar multiplication on a compressed image, it can be performed directly on the quantized coefficients, as long as the QTs of the images are taken into account. Note that if the QTs of both images are proportional as in equation 2-3, equation 2-7 simplifies to

$$H_Q[u,v] = \alpha\gamma F_Q[u,v] \qquad \text{EQ 2-8}$$

The special case γ=1 is where the quality of the input and output images are the same. Note also that if a value in the input, $F_Q[u,v]$, is zero, the corresponding value in the output, $H_Q[u,v]$, is also zero. Thus, this operation can be implemented on an RLE block by simply scaling the values in the data structure - there is no need to reconstruct the quantized array or even the zig-zag vector. When implemented this way, useless multiplies where $F_Q[u,v]$ is zero are avoided. For these reasons, the operation is very fast.

Scalar Addition

Now consider the operation of scalar addition. In this operation, if the value of a pixel in the original image is $f[i,j]$, the value of the corresponding pixel in the output image $h[i,j]$ is given by

$$h[i,j] = f[i,j] + \beta \qquad \text{EQ 2-9}$$

Now, adding a constant to each pixel changes the mean (i.e., the DC) value, which the DCT stores at the [0,0] entry. Thus, only this coefficient should be affected. This can be easily proven using equations 2-1 through 2-4, equation 2-9, and the properties of the DCT [45,37]. The result of such a calculation is

$$H_Q[u,v] = \frac{q_F(u,v)}{q_H(u,v)}F_Q[u,v] + \frac{8\beta}{q_H(u,v)}\delta(u)\delta(v) \qquad \text{EQ 2-10}$$

$$\text{where} \quad \delta(u) = \begin{cases} 1 & u=0 \\ 0 & u \neq 0 \end{cases}$$

If the QTs of both images are proportional as in equation 2-3, equation 2-10 assumes a particularly simple form expressed in the equations:

$$H_Q[0,0] = \gamma F_Q[0,0] + \frac{8\beta}{q_H(0,0)} \qquad \text{EQ 2-11}$$

$$H_Q[u,v] = \gamma F_Q[u,v] \qquad (u,v) \neq (0,0) \qquad \text{EQ 2-12}$$

Again we see that the operation of scalar addition can be performed directly on the quantized coefficients. More importantly, in the common case where γ=1 (i.e., when the quality of the output image is the same as the quality of the input image), this operation involves much less computation than the corresponding operation on uncompressed images, since only the (0,0) coefficient of the quantized matrix is affected.

Pixel Addition

The operation of pixel addition is described by the equation

$$h[i,j] = f[i,j] + g[i,j] \qquad \text{EQ 2-13}$$

As in the case of scalar multiplication, the operation we wish to perform is linear. Since the JPEG compression algorithm is also linear, the quantized coefficients of the output image, $H_Q[u,v]$ are summed, scaled copies of $F_Q[u,v]$ and $G_Q[u,v]$. Specifically, it can be shown, using equations 2-1 through 2-4, equation 2-13, that

$$H_Q[u,v] = \frac{q_F(u,v)}{q_H(u,v)}F_Q[u,v] + \frac{q_G(u,v)}{q_H(u,v)}G_Q[u,v] \qquad \text{EQ 2-14}$$

Once again we see that if the QTs of the images are taken into account, the operation can be performed directly on the quantized coefficients, and that if the QTs for all the images are proportional (with proportionality constants γ_F and γ_G), equation 2-14 can be simplified to:

$$H_Q[u,v] = \gamma_F F_Q[u,v] + \gamma_G G_Q[u,v] \qquad \text{EQ 2-15}$$

Pixel Multiplication

Finally, the operation of pixel multiplication is expressed in the equation

$$h[i,j] = \alpha f[i,j]\, g[i,j] \qquad \text{EQ 2-16}$$

where α is a scalar value. The scalar α, although mathematically superfluous, is convenient to scale pixel values as they are multiplied. This formulation is used, for example, when the image *g* contains pixel values in the range [0..255] but should be interpreted as the range [0..1), as is the case when g is a mask. This operation is realized by setting α to 1/256.

Let $F(v_1, v_2)$, $G(w_1, w_2)$ and $H(u_1, u_2)$ be the quantized values of the compressed images for *f*, *g*, and *h*, respectively. Then using equations 2-1, 2-2 and 2-16, the value of $H(u_1, u_2)$ can be computed as follows:

$$H(u_1, u_2) = \frac{1}{4q_H[u_1, u_2]} \sum_i \sum_j C(i, u_1)\, C(j, u_2)\, h(i, j) \qquad \text{EQ 2-17}$$

$$= \frac{\alpha}{4q_H[u_1, u_2]} \sum_i \sum_j C(i, u_1)\, C(j, u_2) f(i, j)\, g(i, j)$$

$$= \frac{\alpha}{4q_H[u_1, u_2]} \sum_i \sum_j C(i, u_1)\, C(j, u_2)$$

$$\left(\frac{1}{4} \sum_{v_1} \sum_{v_2} C(i, v_1)\, C(j, v_2)\, q_F[v_1, v_2]\, F(v_1, v_2)\right)$$

$$\left(\frac{1}{4} \sum_{w_1} \sum_{w_2} C(i, w_1)\, C(j, w_2)\, q_G[w_1, w_2]\, G(w_1, w_2)\right)$$

$$= \sum_{v_1, v_2, w_1, w_2} F(v_1, v_2)\, G(w_1, w_2)\, W_Q(v_1, v_2, w_1, w_2, u_1, u_2)$$

where

$$W_Q(v_1, v_2, w_1, w_2, u_1, u_2) \qquad \text{EQ 2-18}$$

$$= \frac{\alpha q_F[v_1, v_2]\, q_G[w_1, w_2]}{64 q_H[u_1, u_2]} W(u_1, v_1, w_1)\, W(u_2, v_2, w_2)$$

with

$$W(u, v, w) = \sum_i C(i, u)\, C(i, v)\, C(i, w) \qquad \text{EQ 2-19}$$

This rather lengthy sum can be efficiently computed by noticing several facts:

1. for typical compressed images, $G(w_1, w_2)$ and $F(v_1, v_2)$ are zero for most values of (v_1, v_2) and (w_1, w_2).
2. Of the 256K elements in the function $W_Q(v_1, v_2, w_1, w_2, u_1, u_2)$, only about 4% of the terms are non-zero. In other words, the matrix W_Q is very sparse.

When this method is implemented, care must be taken to evaluate only those terms that might contribute to the sum. Property (1) is used when this method is implemented on RLE blocks, since the zeros are easily skipped. To take advantage of (2), the data structure described in the following paragraphs is used. Since the algorithm operates on RLE blocks, the zig-zag ordered indices are used to reference data elements. By convention, *x, y* and *z* will represent the zig-zag ordered indices of the pairs (v_1,v_2), (w_1,w_2) and (u_1,u_2), respectively. With this substitution, equation 2-17 can be written as

$$H(z) = \sum_{x,y} F(x)\,G(y)\,W_Q(x,y,z) \qquad \text{EQ 2-20}$$

with the summation over *x* and *y* running from 0 to 63.

The following data structure allows equation 2-20 to be compute efficiently. A *combination element* is a pair of numbers z and W, where z is an integer and W is a floating point value. A *combination list* is a list of combination elements. A *combination array*, is a 64 by 64 array of combination lists. The C code shown in figure 2-6 initializes the combination array comb[x,y]. The array contains empty lists when the code is entered. The function ZigZag(u1,u2) returns the zig zag index associated with the element (u_1,u_2). The function AddCombElt (z, W, comb[x,y]) inserts the combination element (z, W) in the combination list stored in the global combination array comb[x,y] (the place of insertion is unimportant) and returns the modified combination list. The array W[8][8][8] is assumed to be initialized with the values of the W function of equation 2-19.

Using the initialized combination array, the C code shown in figure 2-7 efficiently implements equation 2-20 on two RLE blocks `f` and `g`. This algorithm is called the *convolution algorithm*. Notice that `comb[]` is a constant in the code; once computed for the given QTs, it can be applied to an unlimited number of images.

The code operates as follows: The array `hzz`, which represents a zig-zag vector, is assumed to be all zero. For each pair of RLE values in the two input images `f` and `g`, the zig-zag indices `x` and `y` are computed, and the product of

```
ConvolveInit (alpha, fQT, gQT, hQT)
   float alpha, *fQT, *gQT, *hQT;

{
   int u1, u2, v1, v2, w1, w2;
   int x, y, z;
   float t1, t2;

   for (u1=0; u1<8; u1++)
      for (v1=0; v1<8; v1++)
         for (w1=0; w1<8; w1++)
            if ((t1 = W[u1][v1][w1]) != 0.0)
               for (u2=0; u2<8; u2++)
                  for (v2=0; v2<8; v2++)
                     for (w2=0; w2<8; w2++)
                        if ((t2 = W[u2][v2][w2]) != 0.0) {
                          x = ZigZag(u1, u2);
                          y = ZigZag(v1, v2);
                          z = ZigZag(w1, w2);
                          W = t1*t2*alpha*fQT[x]*gQT[y]/hQT[z];
                          comb[x,y] = AddCombElt(z,W,comb[x,y]);
                     }
}
```

Figure 2-6: Initialization of the Combination Array

their data values, which is stored in `tmp`. The z value of each combination element in the combination list stored in `comb[x,y]` is used to determine which elements in the output array `hzz` are affected and accumulate the product `W*tmp` into each element. In this way, only the multiplies that result in non-zero products accumulate in `hzz`. When used in a program, a final pass is needed to run-length encode the zeros, perform integer rounding, and construct the resulting RLE block[2].

Summary of Operations

This section showed how pixel addition, pixel multiplication, scalar addition, and scalar multiplication can be implemented on quantized matrices. As noted

[2] Of course, using integer arithmetic might provide an increase in performance, but I chose to describe the floating point implementation for clarity.

```
Convolve (f, g, hzz)
   RLE_Block *f, *g; /* The input images */
   float *hzz;       /* Array of 64 elements */

{
   int x, y, z;
   float W, tmp;
   RLE_BLOCK *gtmp;
   COMB_LIST *cl;

   for (x=0; f != NULL; f = f->next) {
      x += f->skip;
      for (y=0, gtmp = g; gtmp != NULL; gtmp = gtmp->next) {
         y += gtmp->skip;
         tmp = f->val*gtmp->val;
         for (cl = comb[x,y]; cl != NULL; cl = cl->next) {
            z = cl->z;
            W = cl->W;
            hzz[z] += tmp*W;
         }
      }
   }
}
```

Figure 2-7: Implementation of the Convolve Function

earlier, these transformations can operate directly on RLE blocks. Table 2-1 summarizes the mapping of image operations into operations on RLE blocks. In the table, the symbol $\gamma_{F,H}(x)$ is defined as

$$\gamma_{F,H}(x) \equiv \frac{q_F[x]}{q_H[x]} \equiv \frac{q_F[u,v]}{q_H[u,v]} \qquad \text{EQ 2-21}$$

and the function $Convolve(F, G, \alpha, q_F, q_G, q_H)$ is defined in equation 2-20 and implemented in figures 2-6 and 2-7.

2.3 Applications

Video data is typically transmitted as a sequence of compressed images. While the entropy encoded data cannot be directly manipulated, section 2.2 showed how several operations can be performed on RLE blocks. Referring to fig-

Operation	**Image Space Definition for $h[i,j]$**	**RLE Definition for $H_{ZZ}[x]$**
Scalar Multiplication	$\alpha f[i,j]$	$\alpha\gamma_{F,H}(x)F_{ZZ}[x]$
Scalar Addition	$f[i,j]+\beta$	$\gamma_{F,H}(x)F_{ZZ}[x]+\frac{8\beta\delta(x)}{q_H[0]}$
Pixel Addition	$f[i,j]+g[i,j]$	$\gamma_{F,H}(x)F_{ZZ}[x]+\gamma_{G,H}(x)G_{ZZ}[x]$
Pixel Multiplication	$f[i,j]g[i,j]$	$Convolve(F,G,\alpha,q_F,q_G,q_H)$

Table 2-1: Mapping of Operations

ure 2-5, most of the processing associated with decompressing and compressing an image can be eliminated by entropy decoding the bitstream, performing the operation on the RLE blocks, and entropy coding the result.

The primitive operations in Table 2-1 can be combined to form more powerful operations such as *dissolve* (the simultaneous fade out and fade in of two sequences of images) and subtitling. The implementation of these operations typically involves the computation of an output image that is an algebraic combination of one or more input images. Many examples of such operations are discussed in [61]. One way to perform the combination on a pair of RLE blocks is to use zig-zag vectors as the intermediate representation to compute the expression. For example, to multiply two RLE blocks and add a third, the `Convolve` function of figure 2-7 would be called on the first two RLE blocks, the third RLE block would be added to the zig-zag vector, and then run length encoding and entropy coding steps would be performed on the result. Figure 2-8 graphically depicts our strategy.

The remainder of this section presents two examples that illustrate this strategy and compares the performance of these new algorithms with the brute force algorithm.

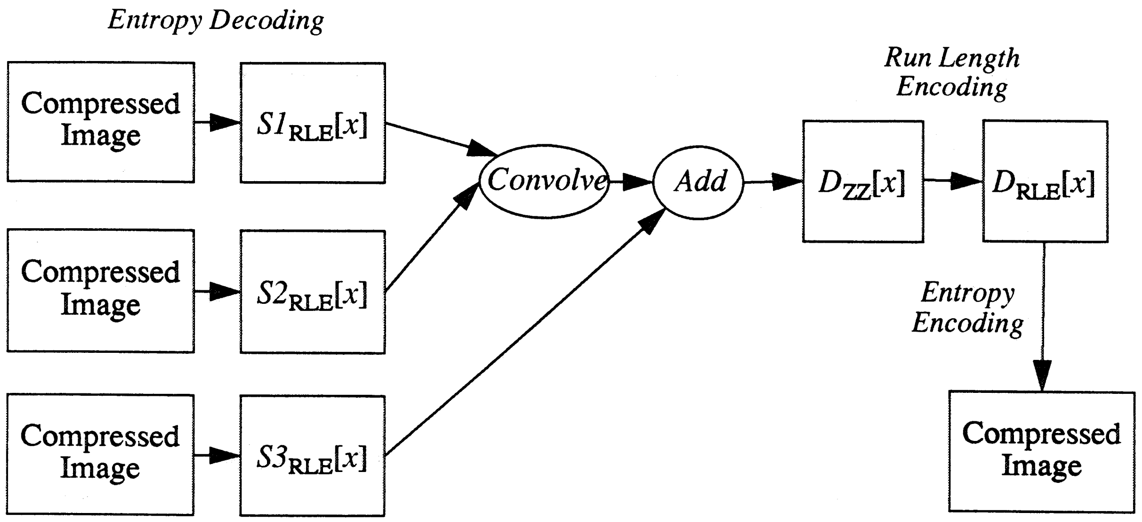


Figure 2-8: Strategy for Manipulating Images

The Dissolve Operation

The entropy encoding and decoding steps will be omitted to simplify the presentation. Suppose a sequence of images S1[t] is to be dissolved into a sequence of images S2[t] in a time Δt (typically 0.25 seconds). In other words, at t=0 S1[0] should be displayed, at t=Δt S2[Δt] should be displayed, and in between a linear combination of the images should be displayed:

$$D[t] = \alpha(t)S1[t] + \{1 - \alpha(t)\}S2[t] \qquad \text{EQ 2-22}$$

where $\alpha(t)$ is a linear function that is 1 at t=0 and 0 at t=Δt.

Using table 2-1, this operation can be mapped into the corresponding operation on RLE blocks as follows. From the table, scalar multiplies can be performed directly on the RLE blocks by changing the coefficient α to $\alpha\gamma_{S1,D}(x)$ in the first half of the expression and performing a similar substitution for the multiplication by $\{1 - \alpha\}$. The table also shows that coefficients can be added together directly to get the desired result, since the QTs of these two new RLE blocks are the same, namely $q_D(x)$. Thus, the expression in equation 2-22 can be implemented as

$$D_{ZZ}[x] = \alpha\gamma_{S1,D}(x)\, S1_{ZZ}[x] + \{1-\alpha\}\,\gamma_{S2,D}(x)\, S2_{ZZ}[x] \qquad \text{EQ 2-23}$$

This equation can be implemented efficiently by noticing that the RLE format of the data will skip over zero terms. The C code to implement this operation on an RLE block is shown in figure 2-9. The function `Zero` zeros the array passed to it, and the function `RunLengthEncode` performs the run length encoding of `h`. The arrays gamma1 and gamma2 have the precomputed values defined by

$$gamma1\,[x] = \alpha\gamma_{S1,D}(x) \qquad \text{EQ 2-24}$$

$$gamma2\,[x] = (1-\alpha)\,\gamma_{S2,D}(x) \qquad \text{EQ 2-25}$$

These values can be precomputed once for each image or sequence of images with the same QTs, whereas the `Dissolve` function is called for each RLE block in an image.

To test the performance of this implementation, I wrote programs that executed both the brute force and the RLE algorithm on images resident in main

```
Dissolve (f, g, h, gamma1, gamma2)
   RLE_Block *f, *g, *h;
   float *gamma1, *gamma2;

{
   float hzz[64];
   int x;

   Zero (hzz);
   for (x=0; f != NULL; f = f->next){
      x += f->skip;
      hzz[x] += gamma1[x]*f->value;
   }
   for (x=0; g != NULL; g = g->next){
      x += g->skip;
      hzz[x] += gamma2[x]*g->value;
   }
   RunLengthEncode (hzz, h);
}
```

Figure 2-9: C Implementation of the Dissolve Operation

memory and compared the performance. Both algorithms were executed on 25 separate pairs of images on a Sparcstation 1+ with 28 MBytes of memory. The test images were 640 X 480, and 24 bits per pixel. The images were compressed to approximately one bit per pixel (24 to 1 compression). Table 2-2 summarizes the results. As can be seen from the table, the speedup is more than 100 to 1 over the brute force algorithm.

Algorithm	Mean Time (sec)	Std Dev (sec)
Brute Force	36.86	0.01
RLE.	0.34	0.00

Table 2-2: Performance Measurements of Dissolve Operation

The Subtitle Operation

The second example operation is *subtitle*, which overlays a subtitle on a compressed image *f*. Although a workstation could support this operation in many ways (such as displaying the text of the subtitle in a separate window), I chose this operation for two reasons. First, it is a common operation that most people understand. And second, it serves as a specific example of the common operation of *image masking*, which is used when a portion of one image is to be combined with another image.

The subtitle is assumed to be a compressed image of white letters on a black background denoted *S*, with white and black represented by pixel values 127 and -128, respectively. The output image can be constructed by adding together *S* and an image obtained by multiplying *f* by a mask that will blacken the areas on *f* where the text should go. The output image with subtitling, *h*, is then given by:

$$h[i,j] = s[i,j] + \frac{1}{255}(127 - s[i,j])f[i,j] \qquad \text{EQ 2-26}$$

Using table 2-1, the corresponding operation on an RLE block is

$$H = \gamma_{s,h} S + Convolve\left(M, F, \frac{1}{255}, q_F, q_S, q_H\right) \qquad \text{EQ 2-27}$$

with

$$M[x] = -S[x] + \frac{1016\delta(x)}{q_S[0]} \qquad \text{EQ 2-28}$$

The C code in figures 2-11 and 2-11 implement this operation. The code is divided into two phases, the `SubtitleInit` function, which is called once when the QTs are defined for the image or sequence of images, and the `Subtitle` function, which is called for each RLE block in the image. Like the dissolve operation, the `Subtitle` function uses a zig-zag vector `hzz` to store the intermediate results. As with the dissolve operation, the performance of programs that implemented the brute force algorithm and the RLE algorithm on images stored in main memory were compared. The test parameters were the same as with the dissolve operation. Table 2-3 summarizes the results, showing a speedup of nearly 50 to 1 over the brute force algorithm.

Algorithm	Mean Time (sec)	Std Dev (sec)
Brute Force	33.84	0.64
RLE	0.68	0.13

Table 2-3: Performance Measurements of Subtitle Operation

```
static gammaSH[64];

SubtitleInit (fQT, sQT, hQT)
   float *fQT, *sQT, *hQT;

{
   int x;

   ConvolveInit (1/255.0, fQT, sQT, hQT);
   for (x=0; x<64; x++)
      gammaSH[x] = sQT[x]/hQT[x];
}
```

Figure 2-10: C Code Implementation of `SubtitleInit` Function

```
Subtitle (s, f, h, fQT, sQT, hQT)
   RLE_Block *s, *f, *h;
   float *fQT, *sQT, *hQT;

{
   float hzz[64];
   RLE_Block *tmp;
   int x;

   Zero (hzz);
   for (x=0, tmp=s; tmp != NULL; tmp = tmp->next){
      x += tmp->skip;
      hzz[x] += gammaSH[x]*tmp->value;     /* hzz = gamma*s */
      tmp->value = -tmp->value;            /* set s = -s */
   }
   /* Convert S into mask... */
   if (s->skip){                           /* No (0,0) value! */
      s = NewRLE_Block(s->next);           /* Insert a 0 value */
      s->skip = 0;
      s->value = 0;
   }
   s->value += Round(1016.0/sQT[0]);       /* s is now the mask */
   Convolve (f, s, hzz);
   RunLengthEncode (hzz, h);
}
```

Figure 2-11: C Code Implementation of `Subtitle` Function

Conclusions

This chapter has examined algorithms for computing local DVEs on compressed JPEG images. This work can be extended in two ways:

1. The ideas can be applied to other compression technologies, such as MPEG, H.261, wavelet, and vector quantization schemes.

2. The class of supported operations can be expanded. For example, non-linear operations can be studied. The next chapter examines one such extension, called *linear global DVEs*.

Chapter 3
Linear Global Digital Video Effects

In the previous chapter, I examined local DVEs, where the value of a pixel in the output image was determined by the value(s) of the corresponding pixel(s) in the input image(s). In this chapter, I study more general DVEs, called global DVEs, where the value of a pixel in an output image is determined by several pixels in an input image. In particular, I examine a set of DVEs called linear global DVEs (LGDVEs), where the value of a pixel in the output image is determined by a linear combination of pixels in the input image. LGDVEs are able to represent many image processing calculations, as well as geometric transformations used in computer graphics, such as translation, scaling, rotation, shearing, and warping. In this chapter, I show how to perform these operation on JPEG data in near real-time. For example, a 640 by 480 JPEG compressed image can be scaled at 7 frames per second on current generation workstations.

The rest of this chapter is organized as follows. Section 3.1 mathematically defines LGDVEs and shows how many DVEs can be written as LGDVEs. In section 3.2, I show how to formulate JPEG compression so that it can be combined (in section 3.3) with the results of section 3.1. The result will be a specification of LGDVEs that operate on compressed data, but is computationally expensive. Section 3.4 develops an approximation technique called *condensation* that exploits properties of the JPEG domain LGDVEs to obtain large performance improvements. In sections 3.5 and 3.6, I describe two condensation algorithms and evaluate their performance. Finally, section 3.7 concludes this work on DVEs with a discussion of the results, a review of related work, and indications of directions for future research in this area.

3.1 The Mathematics of LGDVEs

LGDVEs as Tensors

The class of *linear, global digital video effects* (LGDVEs) are DVEs that can be written in the form

$$h_{uv} = \tau_{uv}^{ij} f_{ij} \qquad \text{EQ 3-1}$$

where τ_{uv}^{ij} is a fourth rank tensor. This notation uses the Einstein summation convention: when an index, such as i, is repeated in the upper and lower indices of two terms, summation over all legal values of the index is implied. For comparison, equation 3-1 could be written in conventional notation as

$$h_{uv} = \sum_{i,j} \tau_{uv}^{ij} f_{ij}$$

LGDVEs, as formulated in equation 3-1, can be used to represent a wide variety of DVEs. As an example, consider the DVE of smoothing an image. In the operation, the value of an output pixel h_{uv} is a weight sum of the nearby pixels in the input image. For example, h_{uv} might be

$$h_{uv} = \frac{1}{2} f_{uv} + \frac{1}{8} f_{u-1,v} + \frac{1}{8} f_{u+1,v} + \frac{1}{8} f_{u,v-1} + \frac{1}{8} f_{u,v+1} \qquad \text{EQ 3-2}$$

Using the definition, equation 3-1, the transform tensor of the corresponding LGDVE is

$$\tau_{uv}^{ij} = \begin{cases} 1/2 & \text{if } i=u \text{ and } j=v \\ 1/8 & \text{if } i=u\pm1 \text{ and } j=v \\ 1/8 & \text{if } i=u \text{ and } j=v\pm1 \\ 0 & \text{otherwise} \end{cases}$$

In fact, convolution of an image with any function $g(x, y)$ can be written as a

LGDVE:

$$\tau_{uv}^{ij} = g(u-i, v-j) \qquad \text{EQ 3-3}$$

LGDVEs can also be used to represent geometric transformations to images. For example, consider the operation of translating an image by an integer amount $\Delta x, \Delta y$. The corresponding LGDVE is

$$\tau_{uv}^{ij} = \delta(i-u+\Delta x, j-v+\Delta y)$$

where $\delta(i, j)$ is the two dimensional Kronecker delta function:

$$\delta(i,j) = \begin{cases} 1 & \text{if } i=0 \text{ and } j=0 \\ 0 & \text{otherwise} \end{cases}$$

The technique for converting a general geometric transform into a LGDVE is discussed in Appendix A.

Although a wide variety of DVEs can be written as LGDVEs, there are many exceptions. Since LGDVEs are inherently linear; no non-linear operations can be formulated as LGDVEs. For example, rotoscoping, which uses severe color quantization for artistic effect, can not be written as an LGDVE.

Block oriented LGDVEs

The LGDVE transformation tensor τ, as specified in the previous section, is specified in terms of the indices of individual pixels in both the input and output images. JPEG encoding, however, is a block oriented technique. That is, to specify an individual pixel, both it's block address (x,y) and it's offset (i,j) within the block are given, as shown in figure 3-1.

It will be convenient to rewrite equation 3-1 using this indexing. By convention, (x,y) will be used for the block address in the input, and (i,j) for the offset within that block. The corresponding symbols for the output address and offset are (w,z)

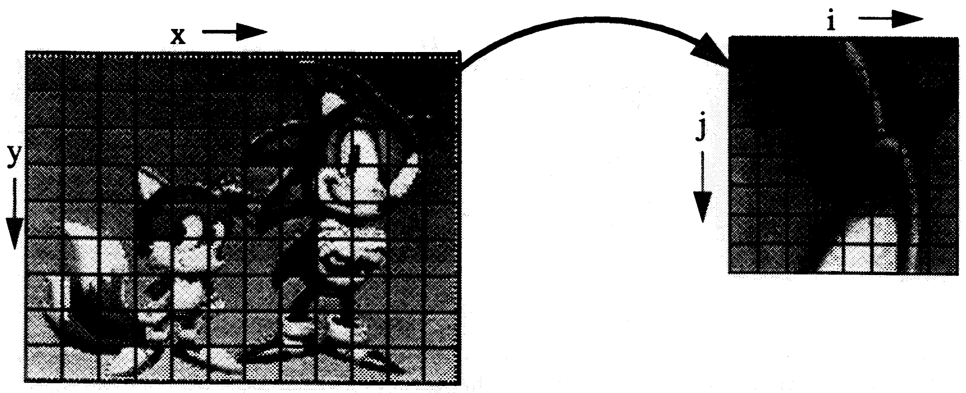


Figure 3-1: JPEG Pixel Addressing

and (u,v), respectively. An LGDVE is then of the form

$$h_{wzuv} = t^{wzuv}_{xyij} f_{xyij} \qquad \text{EQ 3-4}$$

If the blocks are 8x8, $(8x+i,\ 8y+j)$ gives the offsets of a pixel in the image. The correspondence is then

$$\begin{aligned} t^{wzuv}_{xyij} &\approx \tau^{8w+u,\ 8z+v}_{8x+i,\ 8y+j} \\ f_{xyij} &\approx f_{8x+i,\ 8y+j} \\ h_{wzuv} &\approx h_{8w+u,\ 8z+v} \end{aligned} \qquad \text{EQ 3-5}$$

3.2 The Mathematics of JPEG Compression as a Tensor

In the previous section, I showed how certain video effects, namely LGDVEs, could be written as linear operators. In this section I show how several steps in JPEG compression can be written as a linear operator. These two results will be used to derive a method for computing LGDVEs directly on compressed image data.

The JPEG algorithm presented in section 2.1 can be viewed as a four step process. The first step is to apply a linear transformation J^{ij}_{k} to each 8 x 8 pixel block f_{xyij} in the input image, where the subscript xy specifies the index of the

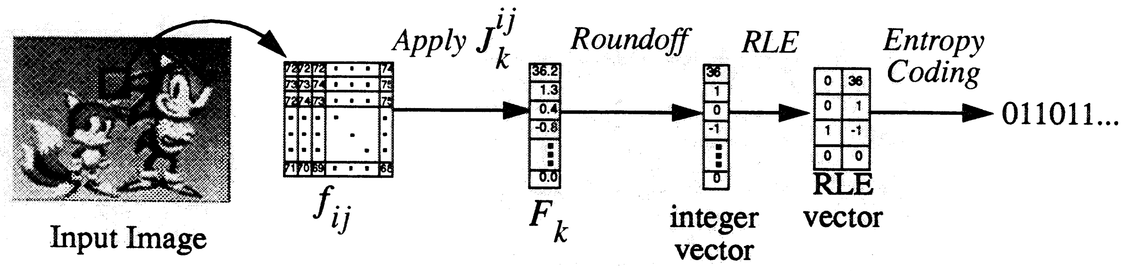


Figure 3-2: The JPEG compression process

block in the input, as shown in figure 3-1. The output of this linear transform is a 64 element vector F_{xyk}. The second step is to round each element of F_{xyk} to the nearest integer, the third step is to produce a sparse vector representation of F_{xyk} using run length encoding, and the final step is to entropy encode this sparse vector. This process is show in figure 3-2.

The linear transformation J_k^{ij} has three steps: a discrete cosine transform (DCT), zig-zag ordering, and division by a quantization value (see section 2.1). These steps define the linear operators D, Z, and Q, respectively, yielding

$$F_{xyk} = \left[Q_k^{\gamma} Z_{\gamma}^{\alpha\beta} D_{\alpha\beta}^{ij}\right] f_{xyij} \qquad \text{EQ 3-6}$$

with

$$Q_k^{\gamma} = \frac{\delta_{\gamma k}}{q[k]} \qquad \text{EQ 3-7}$$

$$Z_{\gamma}^{\alpha\beta} = \begin{cases} 1 & \text{zigzag}[\alpha,\beta]=\gamma \\ 0 & \text{otherwise} \end{cases} \qquad \text{EQ 3-8}$$

$$D_{\alpha\beta}^{ij} = \frac{1}{4} A(\alpha) A(\beta) \cos\frac{(2i+1)\alpha\pi}{16} \cos\frac{(2j+1)\beta\pi}{16} \qquad \text{EQ 3-9}$$

$$A(\alpha) = \begin{cases} \frac{1}{\sqrt{2}} & \alpha=0 \\ 1 & \text{otherwise} \end{cases} \qquad \text{EQ 3-10}$$

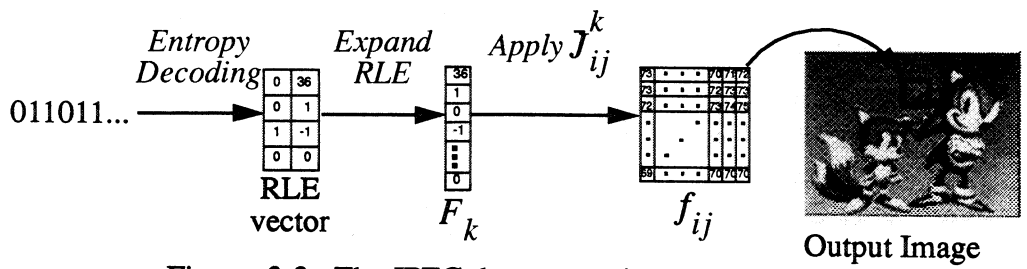


Figure 3-3: The JPEG decompression process

The JPEG operator, J_k^{ij}, is defined as the term in parentheses in equation 3-6:

$$J_k^{ij} \equiv Q_k^{\gamma} Z_{\gamma}^{\alpha\beta} D_{\alpha\beta}^{ij} \qquad \text{EQ 3-11}$$

Decompression of a block is a three part process consisting of entropy decoding, expanding RLE data, and the application of a linear transformation, as shown in figure 3-3.

The linear transformation $\bar{J}$ is the inverse of J, and is defined as

$$\bar{J}_{uv}^{l} = \bar{D}_{uv}^{\alpha\beta} \bar{Z}_{\alpha\beta}^{\gamma} \bar{Q}_{\gamma}^{l} \qquad \text{EQ 3-12}$$

where $\bar{D}$ is the IDCT, $\bar{Q}$ is the dequantizer, and $\bar{Z}$ is the inverse zig-zag operator.

$$\bar{Q}_{\gamma}^{l} = \delta_{\gamma l} q[l] \qquad \text{EQ 3-13}$$

$$\bar{Z}_{\alpha\beta}^{\gamma} = \begin{cases} 1 & \text{zigzag}[\alpha,\beta]=\gamma \\ 0 & \text{otherwise} \end{cases} \qquad \text{EQ 3-14}$$

$$\bar{D}_{uv}^{\alpha\beta} = \frac{1}{4} A(\alpha) A(\beta) \cos\frac{(2u+1)\alpha\pi}{16} \cos\frac{(2v+1)\beta\pi}{16} \qquad \text{EQ 3-15}$$

3.3 Block Oriented LGDVEs on JPEG Data

Having formulated LGDVEs, JPEG compression, and JPEG decompression as linear operators, consider the process of applying an LGDVE on a JPEG compressed image. The following steps need to be performed on the input stream and are shown in figure 3-4:

1. Entropy decode the stream to recover the RLE block representation of F_{xyk}.
2. Expand the RLE block to get F_{xyk}[1]
3. Apply J^{k}_{ij} to F_{xyk} to get f_{xyij}
4. Apply all relevant LGDVE transformation tensors t^{xyij}_{wzuv} to f_{xyij} to get an output block h_{wzuv}
5. Apply J^{uv}_{l} to h_{wzuv} to get H_{wzl}
6. Round off each value in H_{wzl}
7. Run length encode H_{wzl}, and
8. Entropy coding the result.

Steps 3 through 5, can be written as

$$H_{wzl} = \left[J^{uv}_{l} t^{xyij}_{wzuv} J^{k}_{ij}\right] F_{xyk} \qquad \text{EQ 3-16}$$

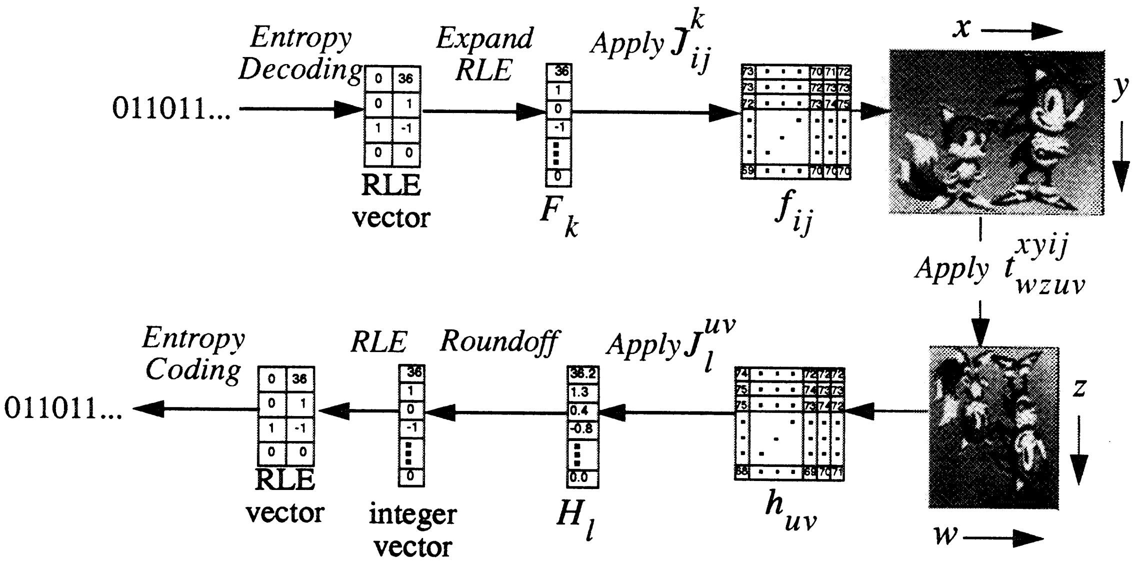


Figure 3-4: Brute force application of LGDVE to JPEG data

[1] This step is optional, since an implementation could apply J directly on the RLE representation.

The term in parentheses is the JPEG representation of the LGDVE tensor t^{xyij}_{wzuv}:

$$T^{xyk}_{wzl} \equiv J^{uv}_{l} t^{xyij}_{wzuv} J^{k}_{ij} \qquad \text{EQ 3-17}$$

T is a linear transform that takes a set of input vectors, F, and produces a set of output vectors, H. It allows the process described in steps 1-8 above to be written in a simpler way:

1. Entropy decode the stream to recover the RLE representation of F_{xyk}.
2. Expand the RLE block to get F_{xyk}
3. Apply T^{xyk}_{wzl} to F_{xyk} get H_{wzl}
4. Round off each value in H_{wzl}
5. Run length encode H_{wzl}, and
6. Entropy coding the result.

This shortcut is illustrated in figure 3-5.

Step 3 is by far the most computationally intensive step in this procedure, and must be implemented carefully. For example, suppose both the source and desti-

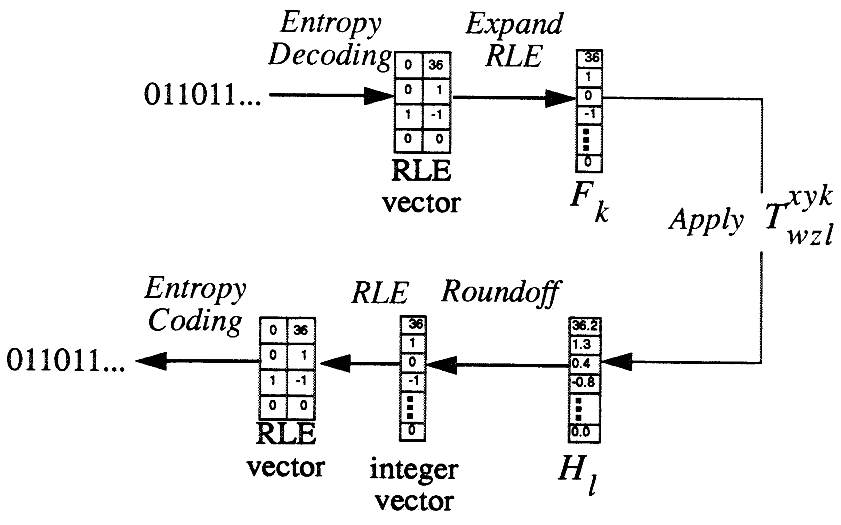


Figure 3-5: Application of JPEG LGDVE to JPEG data

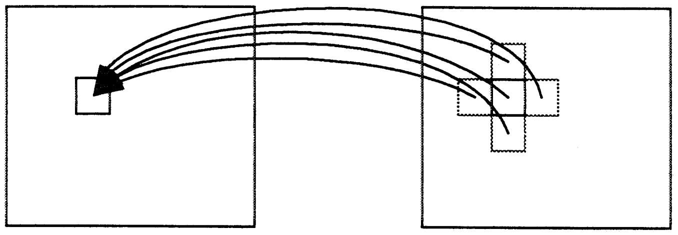


Figure 3-6: Support: the number of input blocks affecting an output block.

nation images are 640 by 480 pixels. The indices x and w can take on 80 different values, y and z can take on 60 different values, and l and k can take on 64 different values. The number of arithmetic operations needed to compute step 3 is then $(80 \times 60 \times 64)^2 \cong 9.4 \times 10^{10}$, which is quite large.

One way to reduce the number of operations is to use the fact that T is typically sparse. To see why T is often sparse, consider the blurring operation defined in equation 3-2. A pixel in the output is determined by the corresponding pixel in the input and it's immediate neighbors. Thus, a block in the output is determined by the corresponding block in the input and it's immediate neighbors, as shown in figure 3-6.

This locality property means that most of T is zero. In the example above

$$T^{xyij}_{wzuv} = 0 \text{ if } |x - w| > 1 \text{ or } |y - z| > 1$$

Thus, for this example, only about 1 in 1000 elements of T are non-zero. Note that most LGDVEs of interest, including convolution filters and geometric transforms of images, exhibit this locality.

In order to measure the sparseness of T, the *support* of a LGDVE tensor is defined as the number of output blocks affected by each input block. For example, the support of the blur filter in equation 3-2 is 5. For 640 by 480 input and output images, the number of operations required to compute the LGDVE is $80 \times 60 \times 64^2 \times N = 1.9N \times 10^7$, where N is the support of the LGDVE tensor.

Although this is three orders of magnitude fewer operations, it is still far too many to compute in real time on today's workstations. The next section explores two techniques that dramatically reduce the number of operations.

3.4 Condensation Algorithms for LGDVEs

This section describes algorithms to condition LGDVEs so that they can be efficiently computed. The conditioning process, called *condensation*, exploits the sparseness of the input blocks and the energy concentration properties of the JPEG tensor (which lead to sparseness in T) to reduce the number of multiplies by a factor of 100 or more without adversely affecting the output quality.

The purpose of condensation is to optimize the computation of a single LGDVE tensor, expressed as a matrix multiply where the 64x64 matrix T multiplies the vector from the input image F to produce the vector from the output image H, as illustrated in figure 3-5. Condensation modifies the tensor T to produce a new tensor T' such that, when T' is applied to an input block F, the resulting block H' will be nearly identical to H. Several properties of the tensor and the input blocks make condensation possible.

Statistical distribution of coefficients in T (Property 1)

Figure 3-7 shows the cumulative distribution of the absolute values of the coefficients of T for the operations of blurring and shrinking. The X axis shows the absolute value of a coefficient, and the Y axis shows the percentage of coefficients with values less than the corresponding X coordinate. Note that T has some elements with very small absolute values for these transformations. For example, the figure indicates that 90% of the coefficients in the shrink-by-2 LGDVE have an absolute value less than 0.05. This property is present in other transforms I investigated. I will discuss the implications of this property shortly.

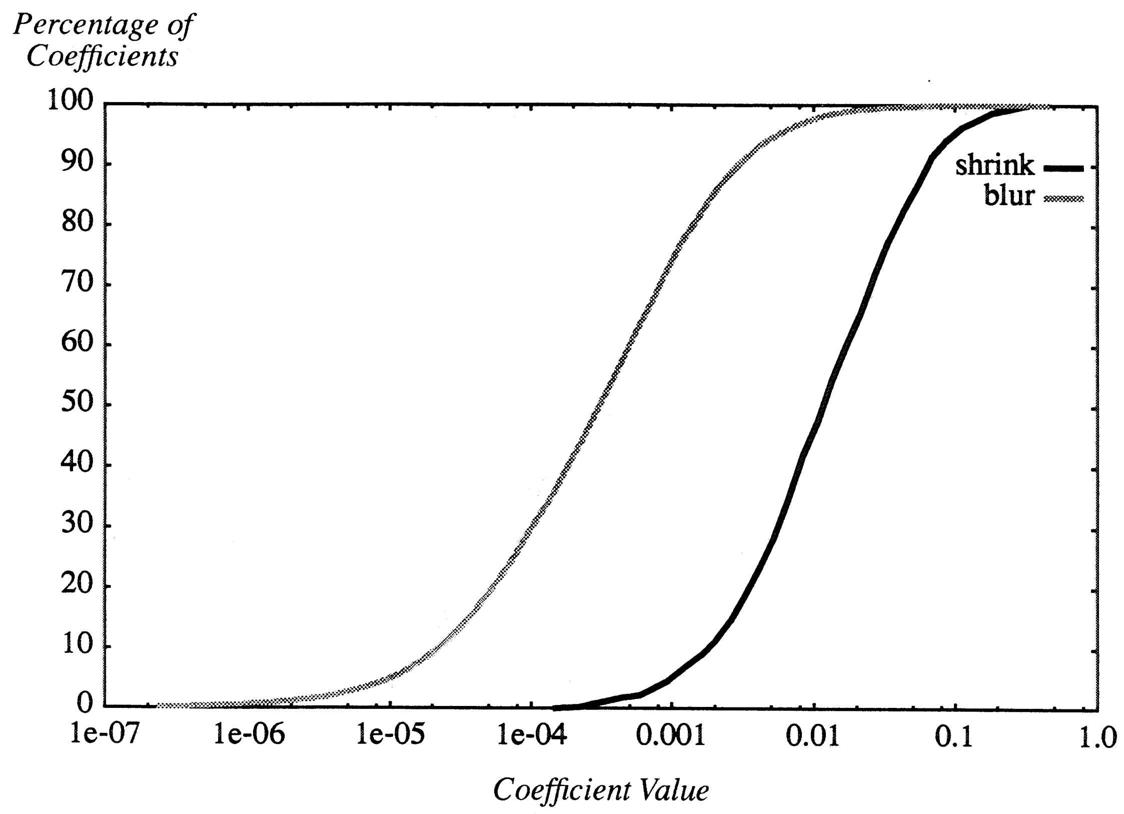


Figure 3-7: Cumulative distributions of absolute values of coefficients for the LGDVEs of blurring and shrinking.

Statistical distribution of input blocks (Property 2)

Figure 3-8 shows the probability distribution $P_k[n]$ of values of input blocks F_k for $k = 10$, a representative value. The data was gathered from 1940 images from 14 categories stored on an FTP archive, compressed using the default quantization tables presented in Annex K of the JPEG draft standard [57]. The X axis indicates the coefficient value, and the Y axis shows the probability that the coefficient has that value. For example, the figure shows that F_{10} has the value 0 most of the time (67%), and a value of -1 only 11% of the time (i.e., $P_3[0] = 0.67$ and $P_3[-1] = 0.11$). Remarkably, these results are nearly independent of the content of the images. Hence, $P_k[n]$ can be empirically computed once and used in any condensation algorithm.

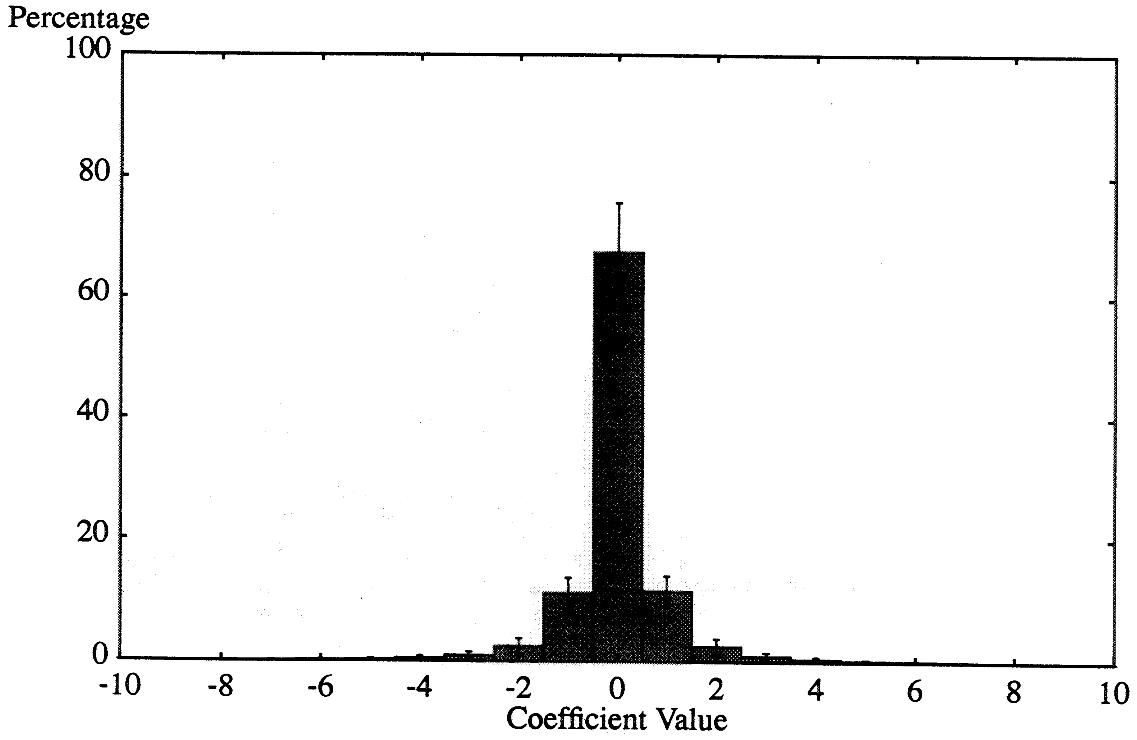


Figure 3-8: Distribution of coefficients in input blocks for k=10

Figure 3-9 is the same as figure 3-8, except that it adds a third dimension of coefficient index. Low values of k (corresponding to low frequencies) are near the front of the graph, higher values are towards the rear. The distributions $P_k[n]$ for the AC coefficients ($k \neq 0$) are peaked at n=0, and become more sharply peaked as k increases. This graph shows that many of the AC coefficients in an input block are zero (i.e., the block is sparse) and the non-zero values are integers close to zero. For the DC coefficient ($k = 0$, not shown) the distribution is nearly flat and highly variable from image to image. The peaking of the AC coefficients is expected, since the DCT tends to concentrate the energy of the image into a few low frequency (small k) coefficients, and the higher frequency components (large k) are more aggressively quantized, leading to more zeros and smaller values in these elements.

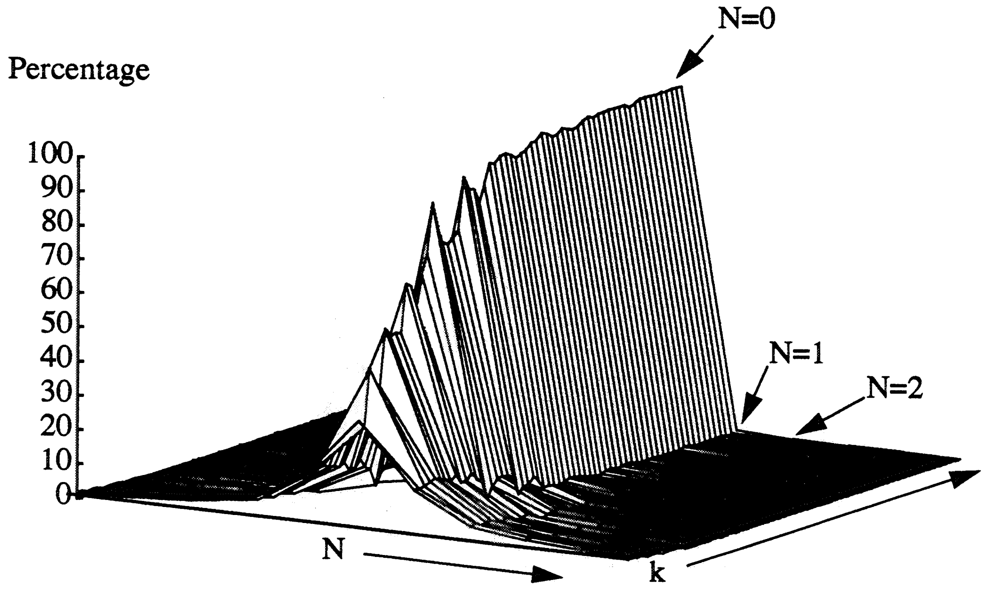


Figure 3-9: Distribution of coefficients in input blocks for AC coefficients

Condensation

Properties 1 and 2 allow an LGDVE tensor T to be conditioned so that it can used more efficiently as follows. For simplicity, consider an LGDVE with a support of one (I'll remove this restriction later). Since the values of F_k are typically small integers (property 2), coefficients in T_l^k with small absolute values, called *insignificant* coefficients, are unlikely to have any impact on the output, and could be set to zero with the introduction of a small error. Such small errors are unlikely to have any effect on a value of H_l, since the real valued sum of $T_l^k F_k$ will be rounded to the nearest integer. So, if the insignificant coefficients are set to zero and T is stored as a sparse matrix, the number of operations required to compute H can be reduced. Since the majority of the coefficients of T are insignificant (property 1), this technique should save a large number of arithmetic operations. The process of setting insignificant coefficients of T to zero is called *condensation.*

I will now formulate the concept of condensation precisely. The matrix T_l^k is used to compute products of the form

$$H_l = T_l^k F_k \qquad \text{EQ 3-18}$$

Now, condensation is the process of finding a set S_l of values of k such that T_l^k can be set to zero for $k \in S_l$. The set S_l is called the *condensation set* for T_l^k.

When T_l^k in equation 3-18 is condensed, an error is introduced in the computation of H_l and the average number of multiples required to compute H_l is lowered. This error is given by

$$E_l = T_l^k F_k - \sum_{k \notin S_l} T_l^k F_k = \sum_{k \in S_l} T_l^k F_k \qquad \text{EQ 3-19}$$

and the expected number of multiplies is given by

$$M_l = \sum_{k \notin S_l} N_k \qquad \text{EQ 3-20}$$

where N_k is the probability that an element F_k is non-zero:

$$N_k \equiv 1 - P_k[0] \qquad \text{EQ 3-21}$$

Unfortunately, each LGDVE tensor T can not be condensed independently when the support is greater than 1. Recall from equation 3-16 that the value of an output block H_{wzl} is computed by applying a group of LGDVE tensors to a group of input blocks:

$$H_{wzl} = T_{wzl}^{xyk} F_{xyk} \qquad \text{EQ 3-22}$$

Now, consider a filter with support 3 and where the values of the T_{12}^{0} elements in the group were 2, -1, and -1. The value of H_{12} is then

$$H_{12} = 2F_0^a - F_0^b - F_0^c$$

where F^a, F^b and F^c are the three relevant input blocks. If the DC coefficients in these three blocks are the same, the terms cancel and the output value H_{12} is zero. Now, suppose a condensation algorithm sets the two -1 values to zero. Then the new value for H_{12} is $H_{12} = 2F_0^a$. Since much of the block's energy in concentrated in the DC component F_0^a, F_0^a is large and the output component H_{12} can be very large. This gives rise to highly visible artifacts in the output. For example, figure 3-10 shows a gray image filtered with a blurring filter where each LGDVE tensor in the filter was condensed independently using the thresholding algorithm described below. Note the pattern of dots in the image. These artifacts are the results of middle frequency coefficients being set to large values.

This situation is remedied by introducing the concept of *bias* in a LGDVE. The bias of an LGDVE is defined as

$$b_{wz}(k, l) = \sum_{xy} T_{wzl}^{xyk} \qquad \text{EQ 3-23}$$

In the example above, $b_{wz}(0, 12)$ is $2 - 1 - 1 = 0$ before condensation,

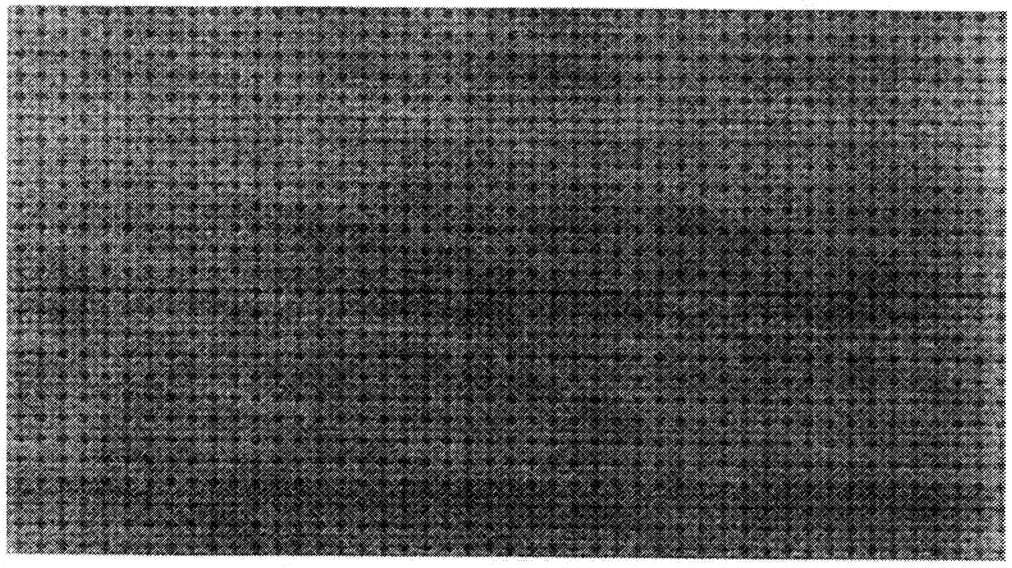


Figure 3-10: Part of a gray image filtered with blur without constant bias

and $b_{wz}(0, 12)$ is 2 after condensation. The change in bias due to condensation caused the visual artifacts in figure 3-10.

These artifacts can be removed by adding the *constant bias* constraint to condensation: the bias of an LGDVE after condensation should be the same as the bias before condensation. To implement the constant bias constraint, the bias of the LGDVE is calculated and stored before applying a condensation algorithm, the LGDVE is condensed, and the remaining non-zero coefficients are adjusted so that the bias is kept constant. More precisely, the steps shown in figure 3-11 are performed.

This code adjusts the coefficients in the LGDVE tensors by distributing the change in $b_{wz}(k, l)$ equally among the non zero coefficients remaining in the LGDVE after condensation. If no coefficients remain, a randomly selected tensor absorbs the change in bias. Figure 3-12 shows a gray image filtered with a blurring filter condensed using the thresholding algorithm, but with constant bias.

The next two sections examine two condensation algorithms (step 2 in figure

```
1 Compute $b(k, l)$ for all (k,l) and all tensors in the support
        of the LGDVE
2 Condense all tensors.
3 Compute $b'(k, l)$ for all (k,l) using the new tensors
4 foreach (k,l) pair do
5    delta = $b(k, l) - b'(k, l)$
6    count := number of tensors with non-zero T[k,l]
7    if (count == 0) then
8         T[k,l] := delta for a randomly selected T.
9    else
10        foreach tensor T with non-zero T[k,l] do
11             T[k,l] := T[k,l] + (delta/count);
12        done
13   fi
14 done
```

Figure 3-11: Bias adjustment

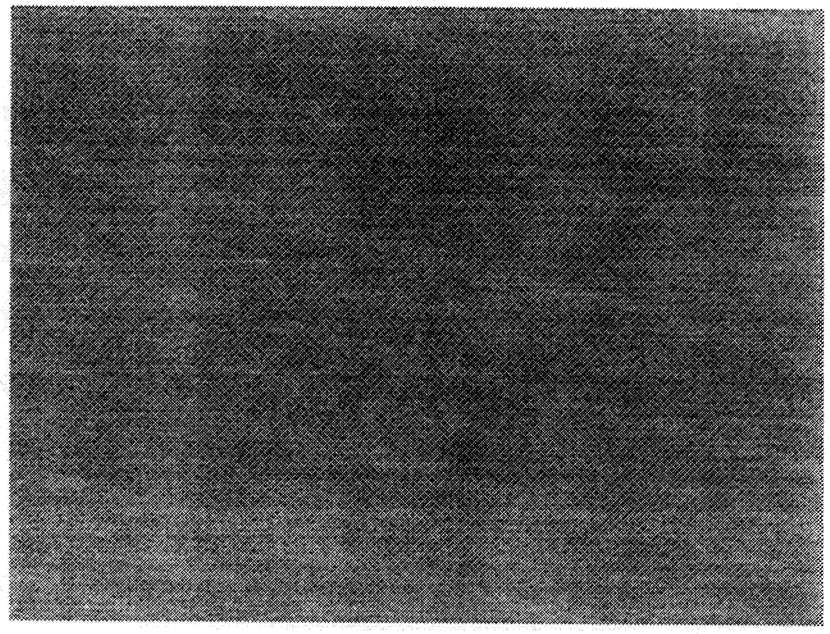


Figure 3-12: Gray image filtered with blur with constant bias

3-11), called *thresholding condensation* and *probabilistic condensation*, and evaluate their performance.

3.5 Thresholding Condensation.

In *thresholding condensation*, a group of matrices T^{xyk}_{wzl} is condensed by distributing a maximum allowable error equally across each term in equation 3-22. For example, suppose an error of 0.5 in the value computed in equation 3-22 can be tolerated. If the support of the filter is one, equation 3-22 has at most 64 terms, Thus, any terms that have a magnitude less than 0.5/64 can be ignored without affecting the result. If the absolute value of any element F_k is never greater than 100, then all elements of T^{xyk}_{wzl} with a value less than 0.5/(64*100) can be set to zero without introducing a cumulative error of more than 0.5 into the sum.

Of course, in the example above the maximum expected value for F_k of 100 was arbitrarily chosen, as was the maximum tolerable error of 0.5. In addition, if the support is N, there are $64N$ terms in equation 3-22. The statistical properties of the input data (property 2 above) can be used to predict a maximum expected value for F_k. If this value is denoted max_k and the maximum tolerable error is called *maxerr*, the condensation set for T_l^k is given by

$$S = \{k, \left|T_k^l\right| < \frac{maxerr}{64 \times N \times max_k}\} \qquad \text{EQ 3-24}$$

Choosing max_k can be done statistically. For example, if max_k is chosen so that no more than a fraction ρ of the values of F_k are greater than this value, then max_k can be computed directly from $P_k[n]$. This leads directly to the algorithm in figure 3-13, called *thresholding condensation.*

The external array P stores the statistical distribution of the coefficients $P_k[n]$, determined off line by examining a large sample of images. In lines 16-20, the maximum expected coefficient value, max_k, is computed and stored in maxK. In lines 21 through 27, any coefficient in the condensation set, as specified by equation 3-24, is set to zero.

Note that the condensation of T can be computed off-line, since it is based on the statistical properties of input images, not on the properties of any individual image. Once computed, the condensed LGDVE can be stored and used when needed.

I implemented this technique to test its performance. The implementation uses straightforward sparse matrix techniques. Given a JPEG stream, all blocks in the stream are decoded and stored as *RLE vectors.* An RLE vector is a structure that stores a sparse vector. It has an array of (index, value) pairs, and a field indicating the size of the array. The sparse input vector F_k is encoded as an array whose elements are (k, F_k) for all values where F_k is not zero. Each LGDVE matrix T_{wzl}^{xyk} is stored as an array of RLE vectors indexed by k that contains the non-zero (l, T_l^k) pairs, as illustrated in figure 3-14.

```
extern float P[64][128];/* P[k][abs(n)] */

Threshold (T, rho, maxErr,N)
    float ***Tptr;              /* Array of N 64x64 matrices */
    float rho;                  /* parameter ρ (see text) */
    float maxErr;               /* Maximum tolerable error */
    int N;                      /* Number of matrices in T */
{
    float sum;
    float **T;
    int i, n, k, l, maxK;

    for (i=0; i<N; i++) {
         T = Tptr[i];
         for (k=0; k<63; k++) {
              sum = 0.0;
              for (n=128; n>=0 && sum+P[k][n]<rho; n--) {
                   sum = sum + P[k][n];
              }
              maxK = n;
              threshold = maxErr/(64*N*maxK);
              for (l=0; l<63; l++) {
                   if (T[l][k] <= threshold) {
                        T[l][k] = 0.0;
                   }
              }
         }
    }
}
```

Figure 3-13: Thresholding Condensation

To compute an output block H, the relevant input blocks and corresponding transformation matrices are found, and each is processed by the loop shown in figure 3-15 and accumulated in the real valued output vector H. After all relevant input blocks have been processed, the vector H is rounded to the nearest integer, and then output as part of the JPEG stream.

Thresholding condensation has two parameters, ρ and *maxerr*, that affect the

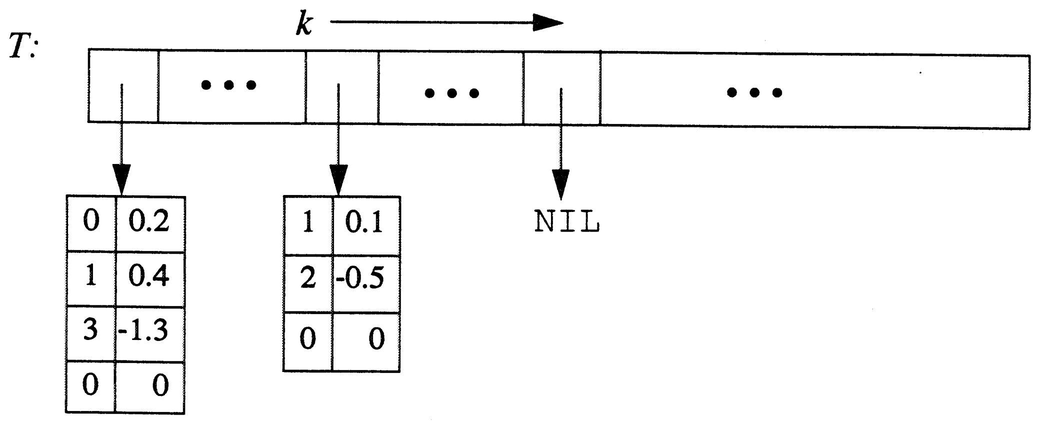


Figure 3-14: Sparse Matrix representation of T

distortion of the output image. I tried several metrics to quantitatively assess the distortion of the output, but found that the state of the art in objective image distortion metrics leaves much to be desired. General purpose metrics such as signal-to-noise ratio (SNR) [37] or modulated SNR [53,65] do not correlate well with subjective results [26]. For example, a gamma corrected image can have the same SNR as an image with overlaid sinusoidal noise, but the gamma corrected image is visually indistinguishable from the original, whereas the sinusoidal noise is very annoying.

Other objective metrics such as those examined in [81,29,86] are designed to detect specific artifacts introduced by compression, and do not detect other artifacts well. For example, the metric proposed in [81] looks for blurring of edges resulting from compression. The errors shown in figure 3-12 go virtually undetected by such a filter.

To measure distortion, I used SNR, defined as

$$SNR = 10\log\left[\frac{rms(O)}{rms(C-O)}\right]$$

where $rms(\bullet)$ is the root mean squared over the image, C is the image calculated using the condensed LGDVE, and O is the image using the original (uncon-

```
typedef sruct ACelem {
   int index, value;
} ACelem;

typedef struct RLE {
   int numAC;
   ACelem ac[63];
} RLE;

ProcessBlock (F, T, H)
   RLE *F;              /* RLE input vector */
   RLE *T[];            /* Array T[k] of RLE vectors */
   float *H;            /* Output array (indexed by l) */
{
   RLE *transform;
   int i, j, k, l;
   float v1, v2;

   for (i=0; i<F->numAC; i++) {
        k = F->ac[i].index;
        v1 = F-ac[i].>value;
        transform = T[k];
        for (j=0; j < transform->numAC; j++) {
             l = transform->ac[j].index;
             v2 = transform->ac[j].value;
             H[l] = H[l] + v1*v2;
        }
   }
}
```

Figure 3-15: Main processing loop for input blocks.

densed) LGDVE.

I tested the implementation on two operators, the *blur* operator (a 7x7 gaussian filter) and the *shrink-by-2* operator, which shrinks an image by a factor of 2 along each dimension. Figure 3-16 shows a graph of the number of multiplies (see eq 3-20) for the blur operation as a function of *maxerr* for several values of ρ, and figure 3-17 shows a graph of the mean SNR for this operation. Figures 3-18

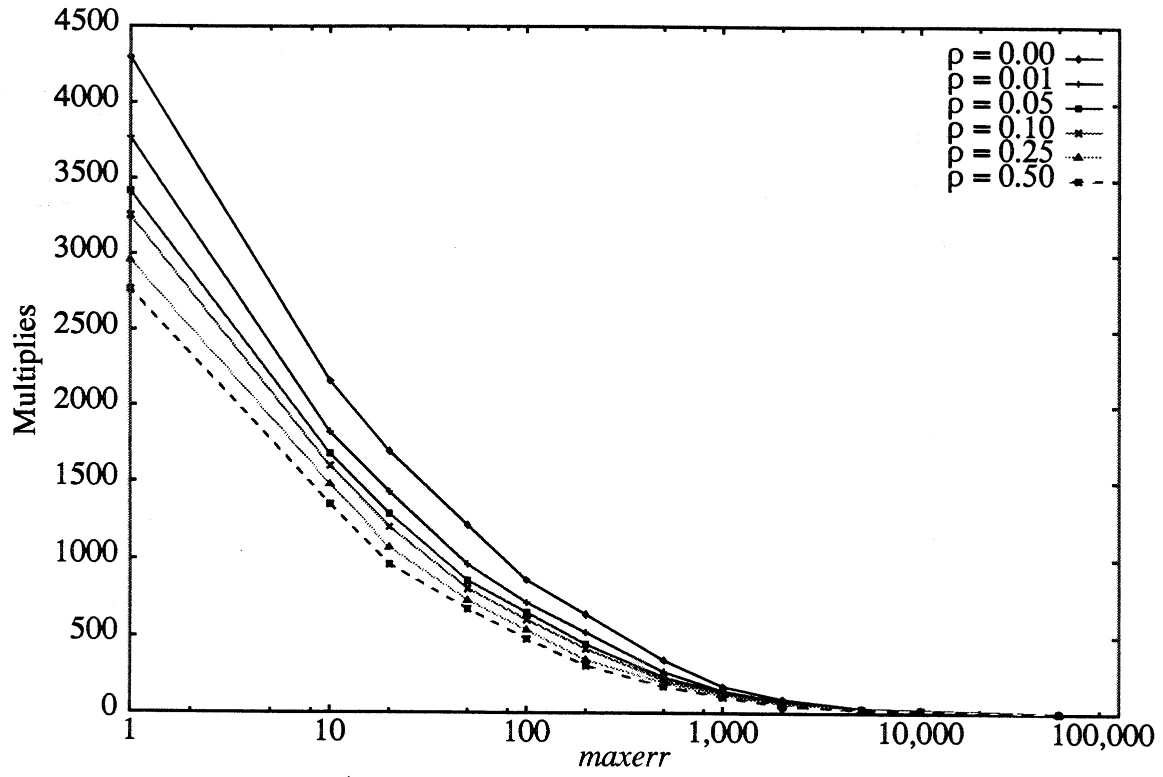


Figure 3-16: Complexity of blur vs. *maxerr* for various values of ρ

and 3-19 are the corresponding graphs for the shrink-by-2 operation. The plots were obtained by applying the thresholding condensation algorithm to the operator at 6 different values of ρ and 12 different values of *maxerr*. The resulting condensed operators were applied to 98 randomly selected grayscale images. The graphs show the average result. With *maxerr*=1, between 40% and 100% of the files (depending on the operator used and the value of ρ) had no error (i.e., SNR = ∞). These data points were left out of the graphs.

Subjective evaluation by the author indicates that at an SNR above about 25, the output quality is quite good, and at an SNR above about 30, the output image is essentially identical to the image computed using the uncondensed operator.

Figures 3-20 and 3-21 show the SNR as a function of the number of multiplies for condensed operators at various values of ρ. These figures show that the SNR is mostly a function of the number of multiplies in the condensed operator, and is

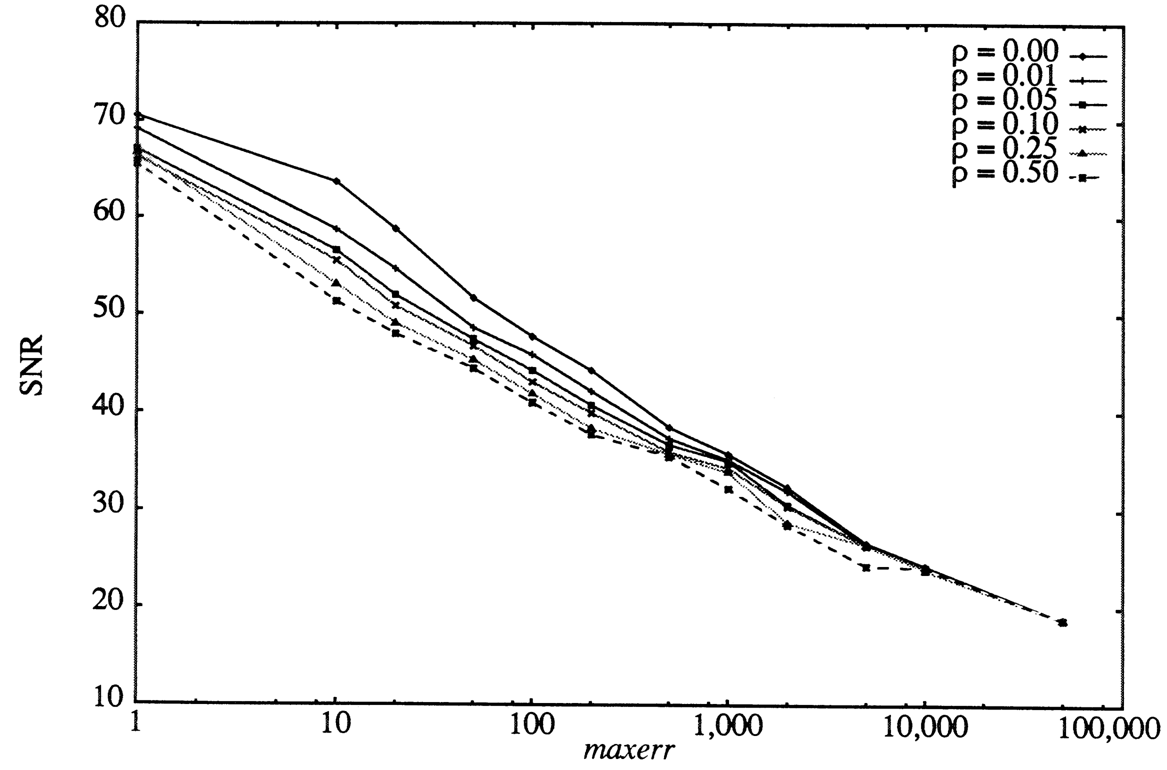


Figure 3-17: SNR of blur vs. *maxerr* for various values of ρ

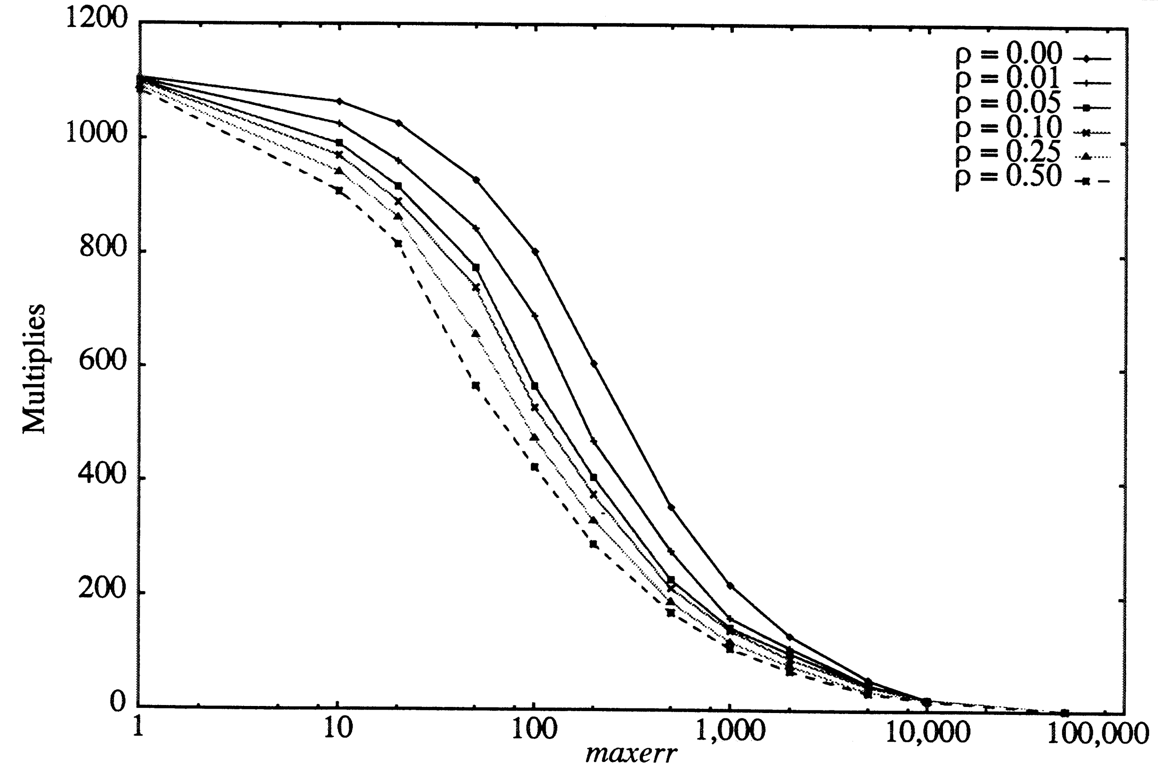


Figure 3-18: Complexity of shrink-by-2 vs. *maxerr* for various values of ρ

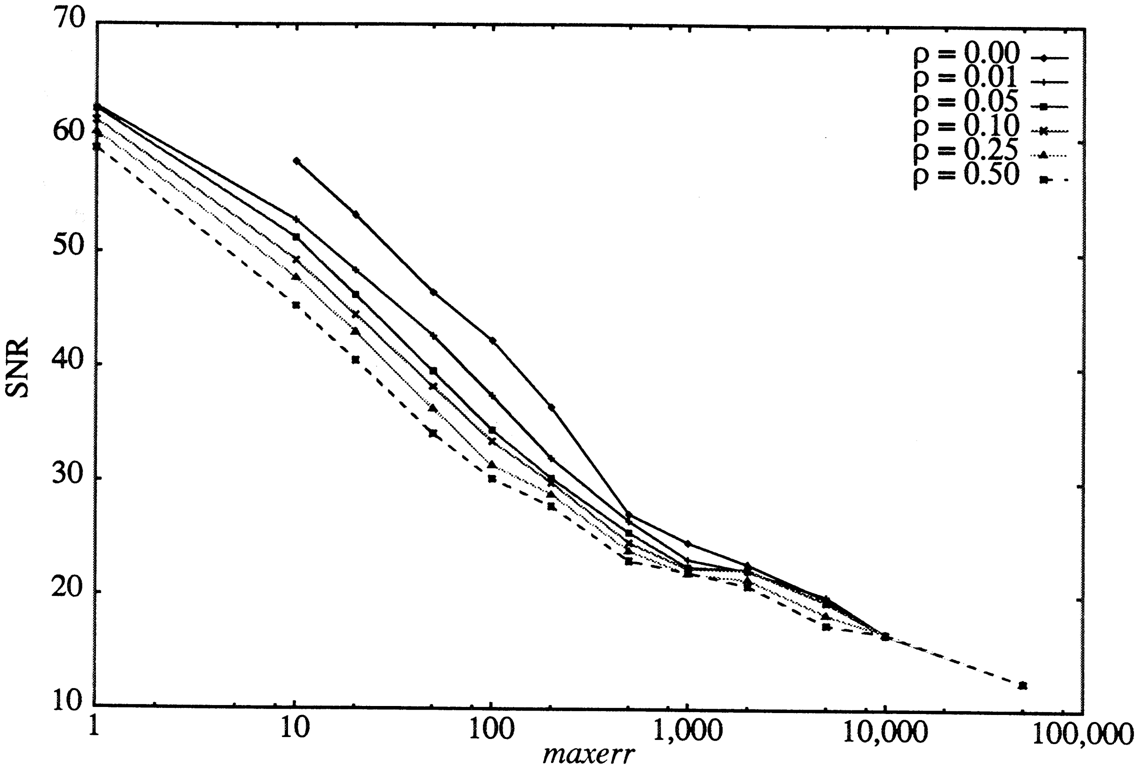


Figure 3-19: SNR of shrink-by-2 vs. *maxerr* for various values of ρ

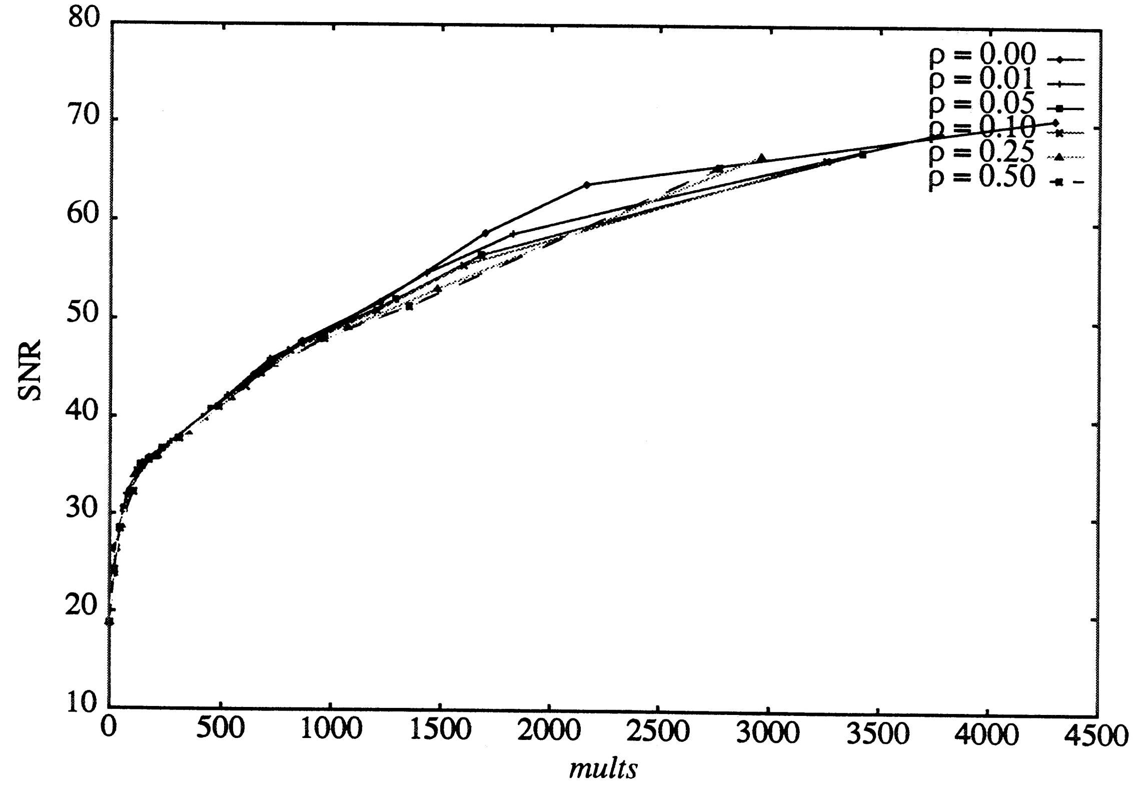


Figure 3-20: SNR of blur vs. number of multiplies for various values of ρ

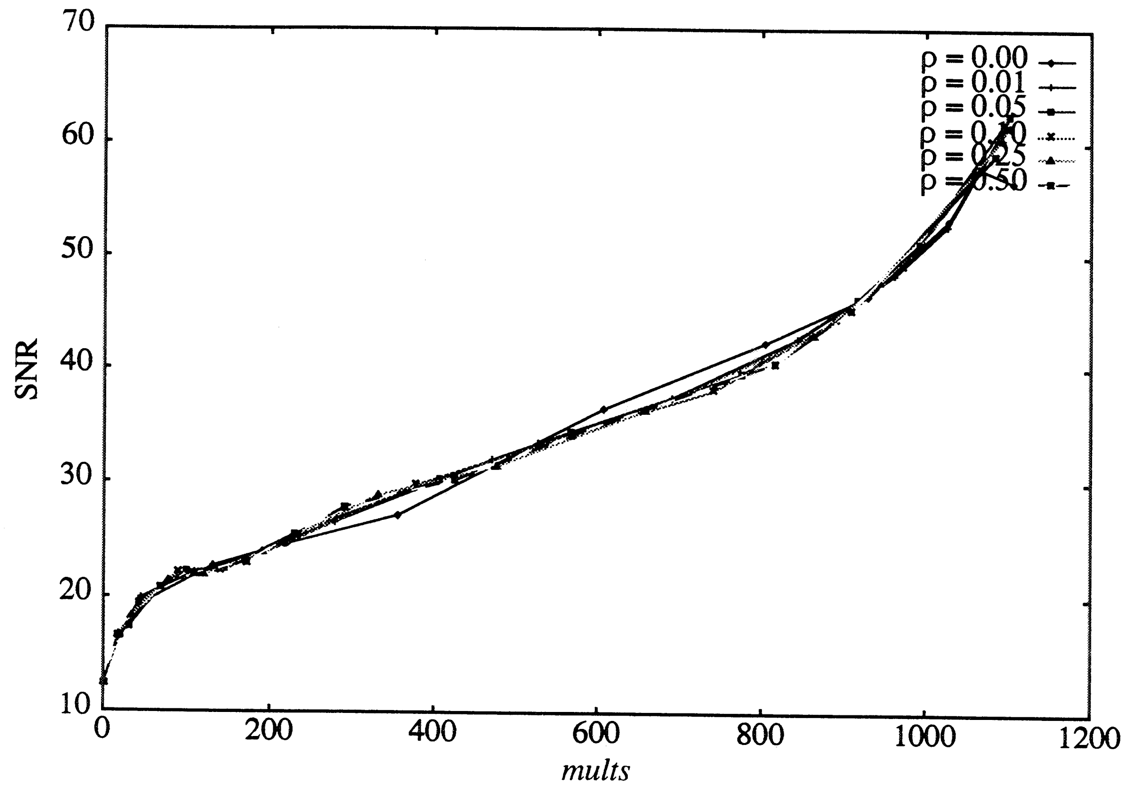


Figure 3-21: SNR of shrink-by-2 vs. number of multiplies for various values of ρ

nearly independent of ρ.

Tables 3-1 and 3-2 compare the performance of the shrink-by-2 and blur operations using condensed operators at various levels of condensation and the brute force method. The experiments were performed on the same suite of 98 images used for my earlier experiments. The tests used a prototype implementation on a DEC 3000/400 workstation with 64 MBytes of memory.

Test Conditions	Time (seconds)	Speedup
SNR = 25	0.290	43.4
SNR = 30	0.331	38.0
Not Condensed	4.45	2.83
Brute Force	12.6	1

Table 3-1: Speed of the blur operation

Test Conditions	Time (seconds)	Speedup
SNR = 25	0.141	5.36
SNR = 30	0.202	3.74
Not Condensed	0.328	2.30
Brute Force	0.755	1

Table 3-2: Speed of the shrink-by-2 operation

Tables 3-3 and 3-4 show an analysis of where time is spent in the program for the two test operators. The Huffman Decoding category includes all phases of reading and decoding the JPEG file into RLE vectors. The Huffman Encoding includes all time involved in encoding the output file, including quantization, run length coding and bitstream generation. The Operator Application phase is the time spent applying the condensed operator as shown in Figure 3-15, and the Overhead category is the time spent in control flow.

Test Conditions	Huffman Decoding	Huffman Encoding	Operator Application	Overhead
SNR = 25	16%	21%	39%	24%
SNR = 30	13%	19%	45%	23%
Not Condensed	1%	2%	95%	2%

Table 3-3: Breakdown of time in computing the blur operation

Test Conditions	Huffman Decoding	Huffman Encoding	Operator Application	Overhead
SNR = 25	47%	20%	23%	10%
SNR = 30	42%	18%	31%	9%
Not Condensed	15%	7%	75%	3%

Table 3-4: Breakdown of time in computing the shrink-by-2 operation

In the blur transformation, less than half the time is spent in application of the condensed operator. For shrink-by-2, only about a quarter of the time is spent in operator application. The rest of the time is spent in overhead and in entropy coding operations. These results indicate that limited performance gains, no more than a factor of 2, are possible by further condensation. Put another way, it takes about as long to entropy code the image as to apply the operator.

3.6 Probabilistic Condensation

Another method for condensation is *probabilistic condensation.* Probabilistic condensation takes a more statistical approach to the problem than thresholding condensation. The error of a condensed matrix, defined in equation 3-19, is a linear combination of the input blocks F_{xyk}. Property 2 (page 40) shows that each F_k has a sharply peaked distributions. By the central limit theorem, the error E_l also has a sharply peaked distribution. In other words, most errors are close to zero, and the chance of large errors is fairly small.

In probabilistic condensation, a condensation set S is constructed that minimizes the expected number of multiplies M, given by equation 3-20, with the constraint that the probability of a significant error is small. Note that since some coefficients F_k are zero more often than others, minimizing the number of multiplies is not the same as making the matrix T_l^k as sparse as possible.

This constraint on the error can be expressed by limiting the width of the distribution of the error term E_l. Variance is a good choice for measuring the width of the distributions F_k, since they are approximately Gaussian and since the central limit theorem states that the distribution of the error term will approach a Gaussian. The variance of E_l is the weighted sum of the variances of F_k:

$$var(E_l) = \sum_{k \in S_l} \left(T_l^k\right)^2 var(F_k) \qquad \text{EQ 3-25}$$

A simple way to compute S_l is to use a cost/benefit approach. The "cost" of adding an element k to S_l is to increase the variance of the error term E_l by $\left(T_l^k\right)^2 var(F_k)$. The "benefit" of adding an element k to S_l is to reduce the expected number of multiplies by N_k (equation 3-21), the probability that a given F_k is non-zero. If B_k is defined as

$$B_k = \frac{N_k}{\left(T_l^k\right)^2 var(F_k)} \qquad \text{EQ 3-26}$$

S_l can be built in order of decreasing B_k, as long the variance in error defined in equation 3-25 is limited to a maximum possible variance, *maxvar*. This strategy leads to the algorithm in figure 3-22.

Lines 1 through 5 of the figure define a record that is used to keep track of the cost/benefit of each of the matrix elements T_l^k. The variance of each F_k, the index k, and the benefit to cost ratio B_k are stored in an array of 64 such structures, allocated at line 13. The procedure `ProbCondense` takes two parameters: an LGDVE tensor `T`, and a limit on the variance in the error term, `maxvar`. Each output element H_l is considered independently. Lines 17 through 25 initialize an array of 64 cost benefit structures, one for each value of k. The array is sorted on B_k, largest elements (i.e., elements that have the most benefit for the least cost) first, at line 26. The elements are then considered in decreasing order of B_k, and the corresponding matrix element T_l^k is set to zero if adding k to the condensation set S_l does not push the variance in error over the specified maximum variance `maxvar`. In the constant bias implementation (figure 3-11), all

```
typedef struct costBenefit {
   float b;                      /* Cost/Benefit (Eq 3-26) */
   int k;                        /* k */
   float variance;               /* variance of $T_l^k F_k$ */
} CostBenefit;

extern float fVar[64];           /* Expected variance in $F_k$ */

ProbCondense (T, maxvar)
   float **T;                    /* 64x64 matrix */
   float maxvar;                 /* Max error variance */
{
   int k,l,i;
   double sumvar;
   CostBenefit cb[64];
   for (l=0; l<64; l++) {
        for (k=0; k<64; k++) {
             cb[k].k = k;
             cb[k].variance = fVar[k]*T[l][k]*T[l][k];
             if (cb[k].variance != 0) {
                  cb[k].b = N[k]/cb[k].variance;
             } else {
                  cb[k].b = 0;
             }
        }
        sort (cb) by b;
        sumvar = 0.0;
        for (i=0; i<64; i++) {
             if (cb[i].variance + sumvar) < maxvar) {
                  k = cb[i].k;
                  T[l][k] = 0.0;
                  sumvar = sumvar + cb[i].variance;
             }
        }
   }
}
```

Figure 3-22: Probabilistic condensation algorithm

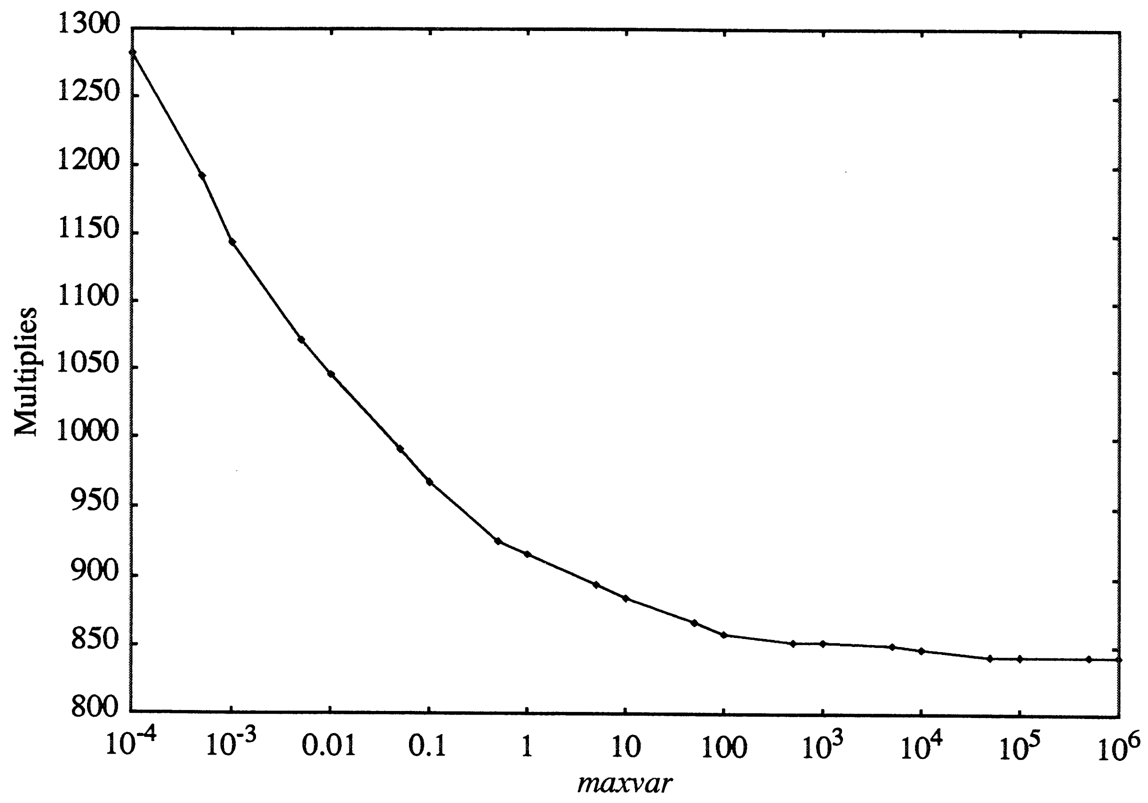


Figure 3-23: Complexity of blur vs. maxvar

matrices in the support of the LGDVE are condensed at once, using a larger `CostBenefit` structure.

In contrast to the thresholding algorithm presented in the previous section, only a single parameter, namely `maxvar`, controls the condensation. Figure 3-23 shows a graph of the computational speedup (given by equation 3-20) for the blur operation as a function of `maxvar`, and figure 3-24 shows the corresponding graph of the SNR. Figures 3-25 and 3-26 graph the complexity and SNR of the shrink-by-2 operation under probabilistic condensation. The plots were obtained under the same experimental conditions used in the thresholding experiments.

Finally, figures 3-27 and 3-28 compare thresholding and probabilistic condensation. From the figures it is clear that in all cases, thresholding condensation gives better results than probabilistic condensation for a given complexity.

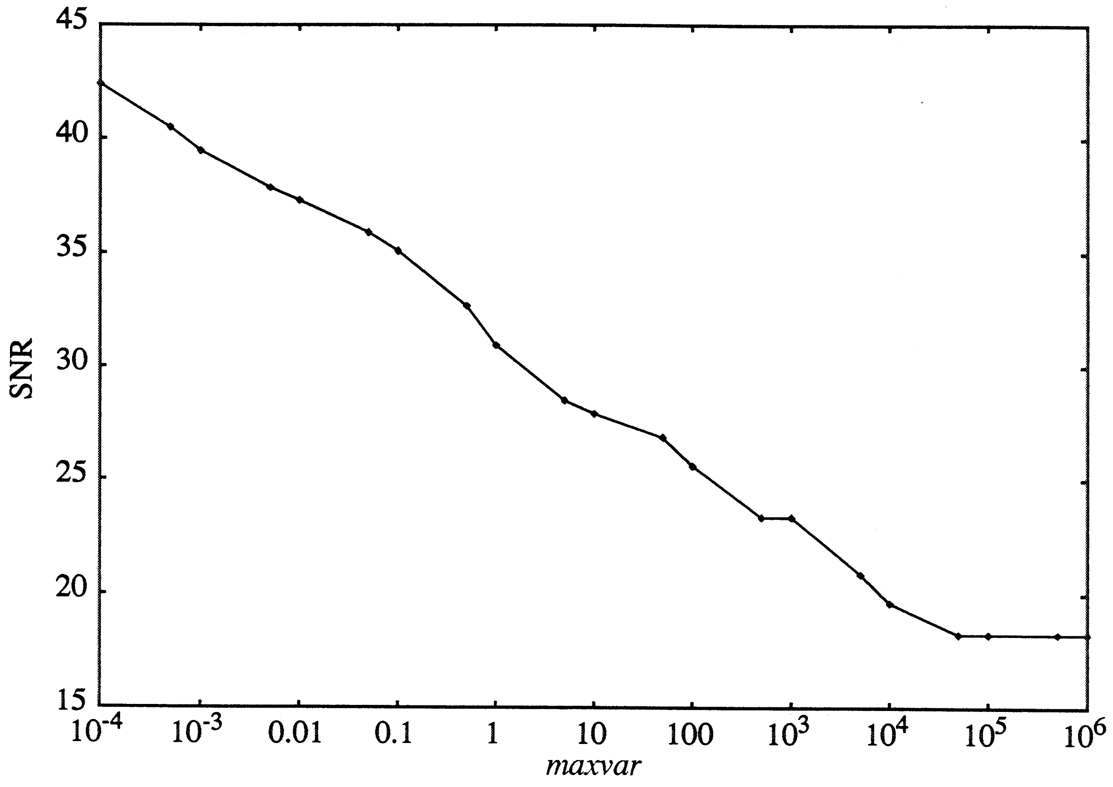


Figure 3-24: SNR of blur vs. maxvar

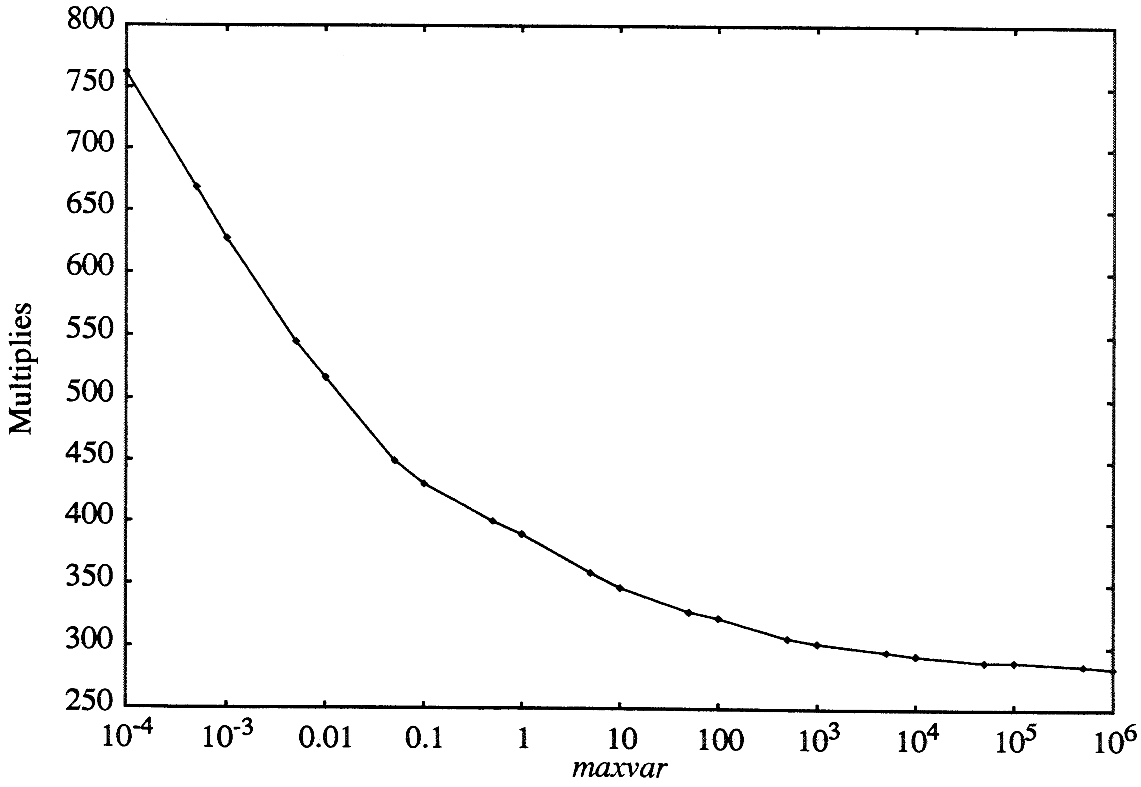


Figure 3-25: Complexity shrink-by-2 of vs. maxvar

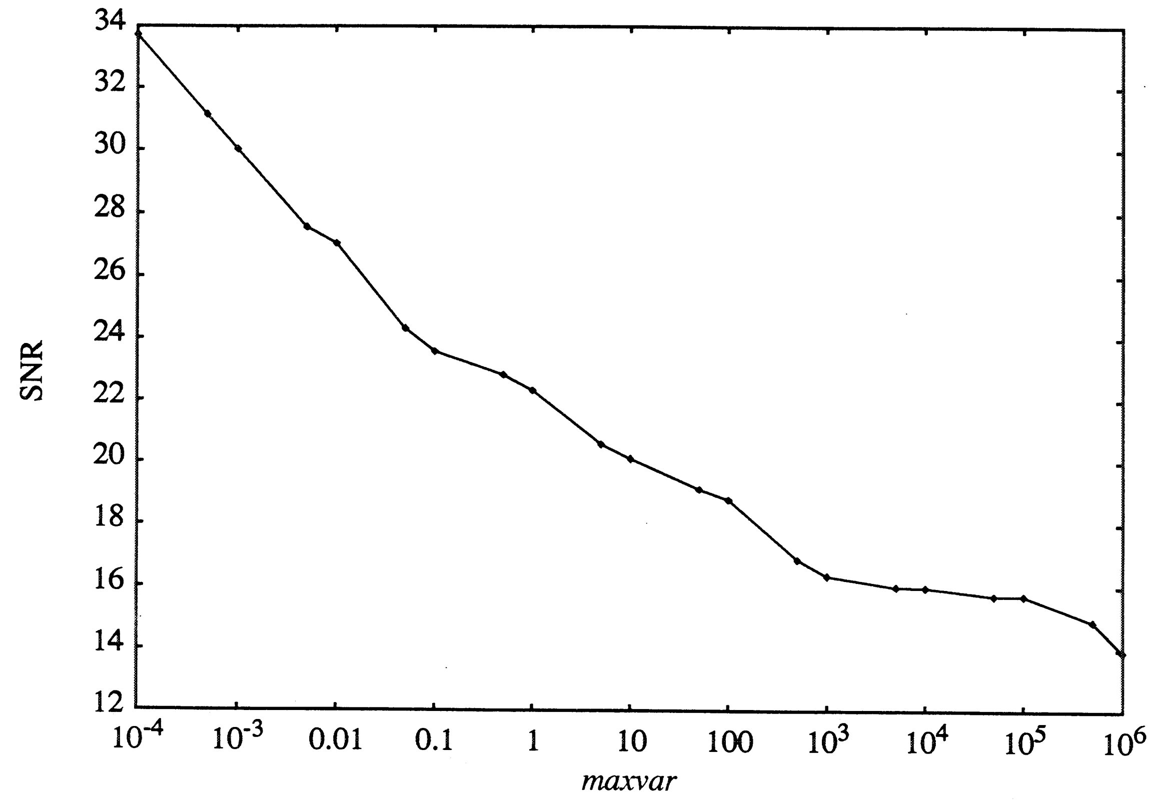


Figure 3-26: SNR of shrink-by-2 vs. maxvar

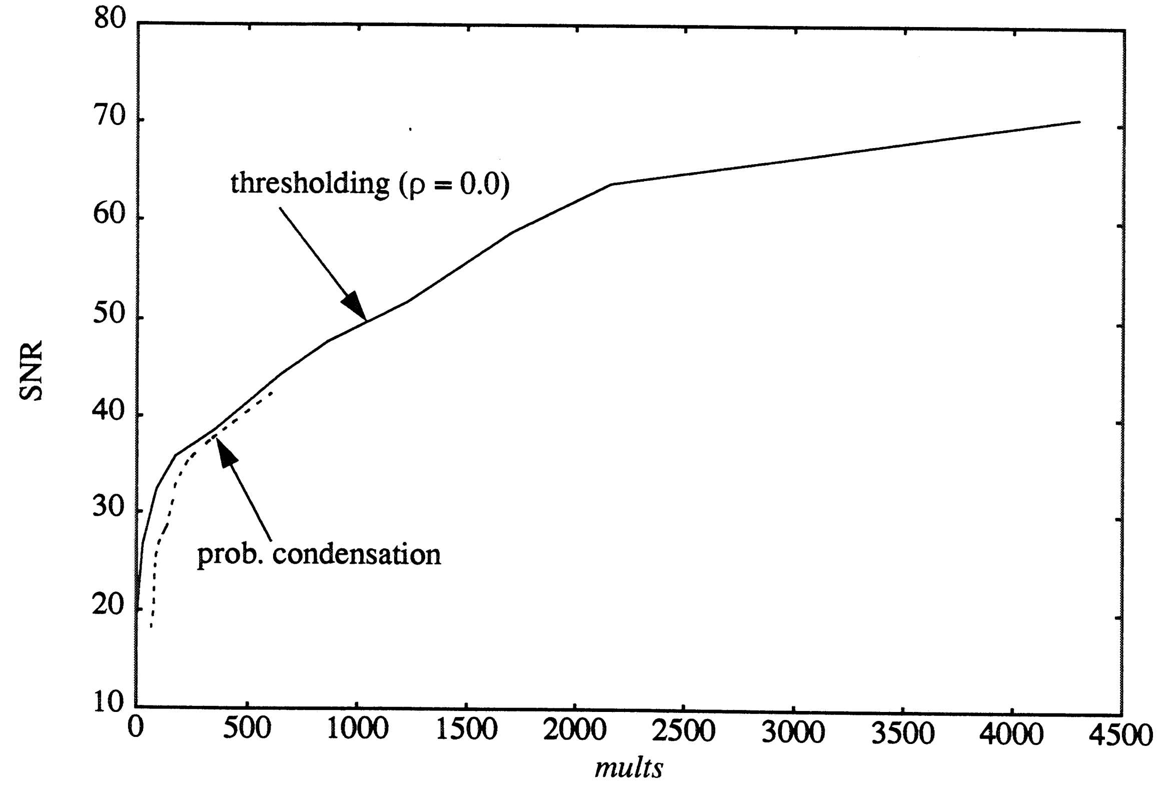


Figure 3-27: Comparison of thresholding and probabilistic condensation for blur

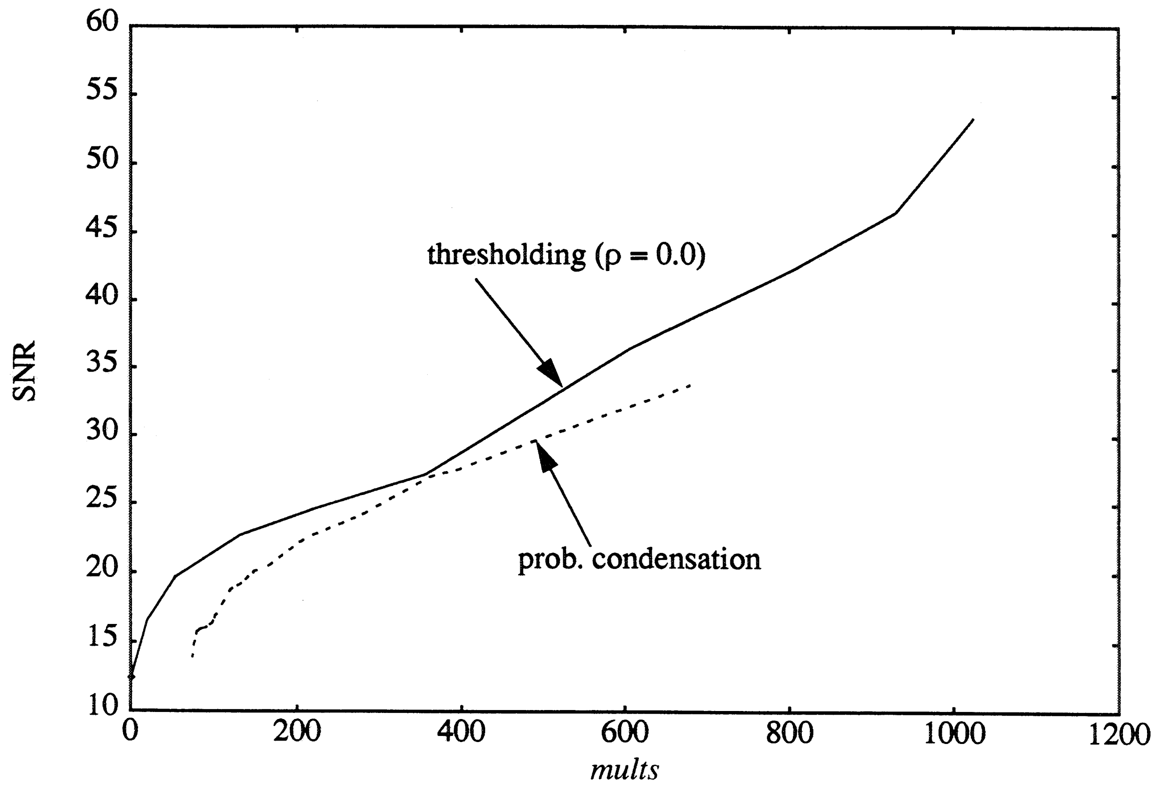


Figure 3-28: Comparison of thresholding and probabilistic condensation shrink-by-2

3.7 Previous Work, Future Work, and Conclusions

This chapter has examined techniques for computing LGDVEs on JPEG data. The techniques can be used to perform a large set of video effects on motion JPEG video data in near real-time on off-the-shelf hardware. For example, the shrink-by-2 operation can be performed at about 7 frames/second on 640 by 480 input images in the test implementation. Tests show the algorithms scale linearly with the input data size, so the shrink-by-2 operation could be performed at about 28 frames/second on VHS quality video input (320 by 240 resolution).

The techniques in this chapter can be made faster by exploiting other properties of the transforms, such as symmetry. For example, when input blocks play a symmetric role in the value of an output block during the application of the LGDVE operator (e.g., in shrink-by-2, the four input blocks have such a role), the coefficients T_{wzl}^{xyk} will have the same absolute value for a fixed k, l. For example, in the shrink-by-2 operations, four input blocks determine the value of a single output block. The DC value of the output block is the average of the DC values of the four input blocks:

$$H_0 = \frac{1}{4}[F1_0 + F2_0 + F3_0 + F4_0]$$

This operation could be computed more efficiently by factoring the equation. That is, sum the D.C. coefficients and then multiply by 0.25. Similarly, factoring is possible in the calculation of the first AC coefficient:

$$H_1 = 0.33\,[F1_0 - F2_0 + F3_0 - F4_0] + 0.1\,[F1_0 + F2_0 + F3_0 + F4_0]$$

In fact, factoring is possible with nearly all the coefficients in shrink-by-2, which is a direct consequence of the symmetry in the input blocks. Similar factoring is possible with the LGDVE implementation of most FIR filters.

Another way to speed up the process is to develop better condensation algorithms. Thresholding condensation uses a very simple technique to bound the error. A better method would exploit the same properties of human vision that are used in JPEG compression. For example, more error can be tolerated in the high frequency coefficients than in the low frequency coefficients. Exploiting masking effects may also be possible.

A third line of future work is to explore the idea of progressive algorithms. In many applications, the time to complete the computation of an LGDVE is fixed (e.g., 33 milliseconds in 30 frames per second video). In such situations, the goal is to compute the best possible output in limited time -- should more compute cycles become available, the output should improve. A progressive implementation of the ideas presented in this chapter would compute the visually important information first and produce progressively higher quality output given more time.

An interesting research question is to explore how the ideas in this chapter can be extended to other compression techniques. A related question is whether a compression standard can be developed that facilitates the application of video effects. Since about 50% of the time spent in computing an LGDVE on JPEG data goes into entropy coding, perhaps techniques that have faster implementations (e.g., Lempel-Ziv encoding [87]) should be used for the entropy coding step.

Finally, system work is important to make these techniques practical. Storage, retrieval, indexing, and data structures dramatically effect the performance of an implementation.

Little work has been done in the area of computing video effects on compressed image data. Chang, et. al. [16] has developed a technique for block alignment of compressed video data, so that local DVEs such as compositing can be applied. In the context of this chapter, this operation is an LGDVE that translates an image. In later work [17], he extends his block alignment techniques to MPEG data.

Duff and Porter [61] describe an algebra for image composition based on the introduction of an α channel. In their method, the source image and a real valued image, the α *channel*, are pixel-wise multiplied before being composited. The results of sections 2.2 and 2.3 of the previous chapter show how two compressed images can be pixel-wise multiplied and composited, which provides a basis for using α channel techniques on compressed images.

In the area of DCT domain image processing, Chitprasert and Rao [18] presented a convolution algorithm for the DCT, and showed how it could be used for image processing in certain special cases. Their technique is applicable to high and low pass filtering, but does not adapt well to the block by block encoding nature of most compression technologies.

Lastly, Arman [4] has developed a technique for detecting scene breaks in motion JPEG video data that operates on the compressed representation. His technique compares two images using the dot product of two vectors formed from the DCT coefficients of a selected subset of corresponding block in the test images. If the dot product exceeds a threshold value, a scene break is declared.

Chapter 4

CMT: A Continuous Media Toolkit

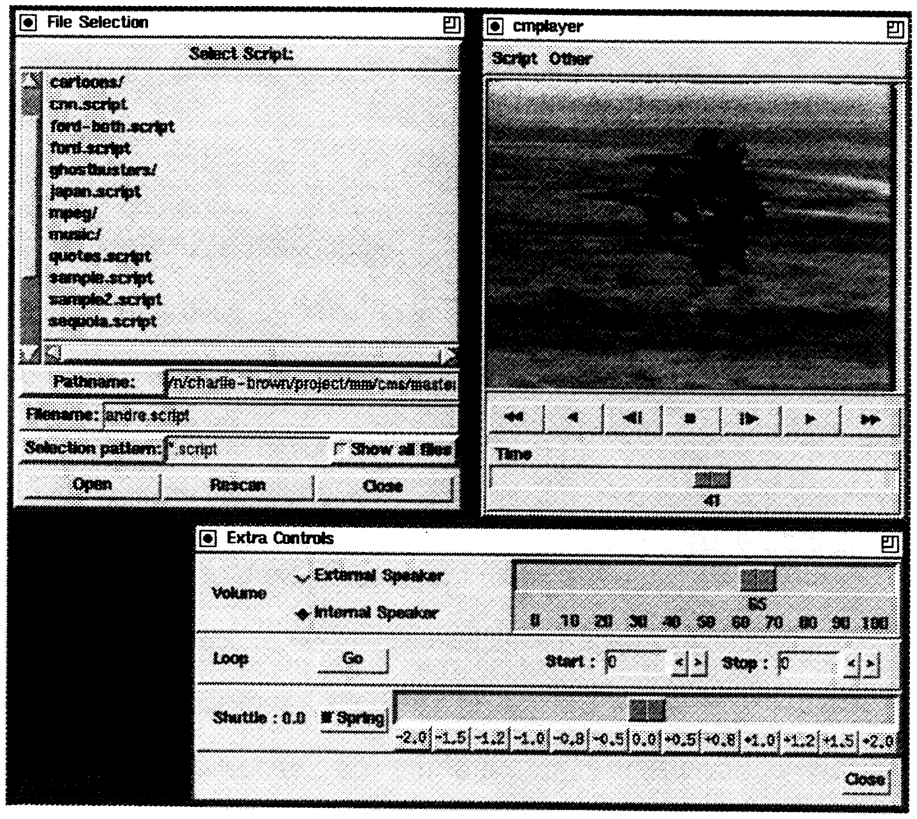
4.1 Introduction

This chapter describes the concepts, data structures, and operations in the Berkeley CM Toolkit (CMT). CMT was used to build a video playback application, called the *CM Player*, that plays audio and video data stored on a magnetic disk. The audio and video data can be stored locally (i.e., on the same machine) or remotely (i.e., connected to the user's machine by a network).

Figure 4-1 shows the user interface to the CM Player. The window on the left lists the available movies; double-clicking on a movie selects it for playback. Video appears in the upper portion of the right hand window, also known as the *main* window. Buttons at the bottom of the main window provide familiar VCR controls: play forward and reverse, stop, fast forward and backward, and single frame step forward and backward. Finally, the slider below these buttons allows direct access to any position in the video.

Advanced controls are provided in the *extra controls* panel, shown below the main window in figure 4-1. The slider at the top of the panel allows the user to set the volume of the audio output, and the radio buttons to the left direct the output to the internal speaker or an external audio device such as a headset or speaker. The lower slider in the extras panel provides a jog/shuttle facility that allows the user to play the movie forward or backward at an arbitrary speed from one-tenth to fifty times normal speed, and the slider buttons allow the user to play the movie at one of several predefined speeds. The user may use any control, in any combination, at any time during playback.

Behind this user interface is a distributed application consisting of three cooperating processes that manage a producer/consumer model for CM data. The CM



cartoons/
cnn.script
ford-beth.script
ford.script
ghostbusters/
japan.script
mpeg/
music/
quotes.script
sample.script
sample2.script
sequoia.script

Pathname: /n/charlie-brown/project/mm/cms/maste

Filename: andre.script

Selection pattern: *.script

Start: 0

Stop: 0

Figure 4-1: CM Player user interface.

Source (CMS) process is a producer process that reads audio and video data from a local disk and sends it to a consumer process, called CMX, which is an abbreviation for *CM Server for X-Windows.* A third process, called the Application, controls communication between CMS and CMX in response to user input. Figure 4-2 shows the process architecture for the CM Player application. Since inter-process communication in CMT is network transparent, the Application and CMS processes can be located anywhere on the network.

Each process in the CM Player serves a specific function. The Application process creates the user interface and handles user input. CMX synchronizes the playback of audio and video packets sent by CMS. And, CMS is essentially a file server; it transmits one or more requested streams of CM data to CMX for display. It differs from conventional file servers in that data is read and sent to CMX based

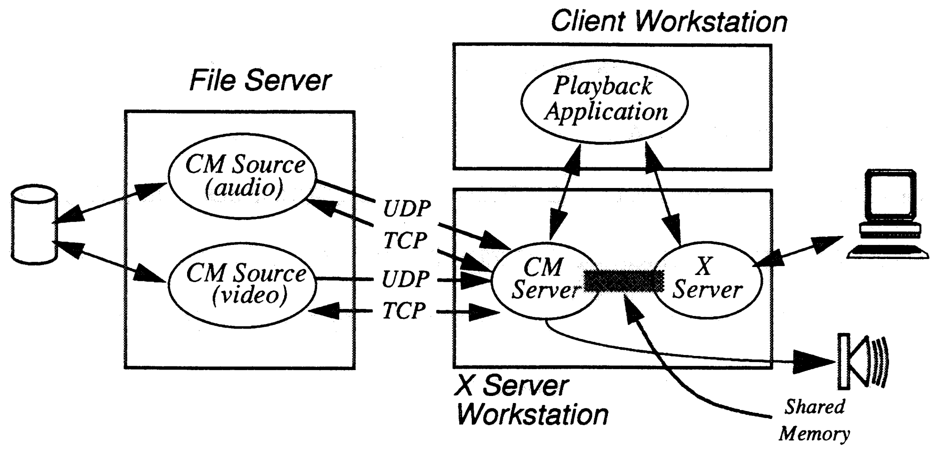


Figure 4-2: CM Player process architecture.

on a schedule, not in response to specific application requests. Data is sent to CMX a short time (typically 0.5 seconds) before it is needed and buffered to reduce the effect of network delays.

CMT allows a movie to be composed from files striped across different CMSs. Striping has several advantages:

1. It allows large video sequences to be split into many small chunks for easier management.
2. It gives greater flexibility when CMSs are used as caches for data stored on a tertiary archive [21, 47].
3. It allows segments to be reused and shared, saving valuable disk storage.

CMT is highly portable, since it is built using generic Unix operating system facilities, conventional networks (i.e., any network supporting TCP/IP and UDP/IP protocols), and off-the-shelf hardware. CMT shows that high quality video playback applications can be built using generic components, which contradicts the view that new network protocols, operating systems, and special purpose hardware are needed for high quality playback. The current system has been ported to DEC, HP, PC, SGI, and Sun platforms.

CMT plays synchronized audio and video. Synchronization is maintained even on unreliable networks with significant network contention. The system uses two custom designed connection protocols: 1) a control connection based on TCP/IP and 2) a data transmission connection based on UDP/IP [75]. The protocol dynamically adjusts the video frames per second (*fps*) sent by CMS to achieve the highest perceived quality of playback given the available network and computing resources.

Experiments show that the system performs well on both local area networks (LANs) and wide area networks (WANs). In LAN experiments, a 2.8 Mbit/sec video sequence with 320 by 240 full color, JPEG compressed images was played at 40 fps when sent across a conventional 10 Mbit ethernet. WAN performance is also impressive. A 352 by 240 MPEG video sequence sent across an eighteen hop Internet route from UC Berkeley to Cornell University was played at over 17 fps. These experiments are described in more detail in the next chapter.

CMT currently includes support for software decoding of MPEG compressed video streams [44], Sun audio files, and hardware decoding of motion JPEG video streams [57,82] using the Parallax XVideo hardware [56]. The code is written in the C programming language [39] and an extended version of the Tcl/Tk language [54,55], which is described below. CMT is composed of approximately 3100 lines in Tcl and 130,000 lines of code in C, 90,000 of which implemented the Tcl/Tk language. Table 4-1 shows the number of lines of C code in each part of the system.

The rest of this chapter describes the architecture of CMT and the CM Player application. Section 4.2 describes the data model used to store CM data. Section 4.3 describes the implementation of the CM Player application using a distributed object system. Finally, section 4.4 compares the system with previous work.

Component	Lines
CMS	6000
CMX	9600
MPEG Decoder	17200
Tcl	22300
Tk	66700
Other	7800

Table 4-1: Breakdown of code in CMT

4.2 Data Model

CMT uses a *storyboard* model to represent the playback schedule of stored CM data. In a storyboard, *streams* of CM data are specified along a timeline. For example, figure 4-2 shows a storyboard with two streams: a video stream and an audio stream. The horizontal axis of the storyboard's coordinate system is called the *logical time system* (LTS). The origin of the LTS (time=0) coincides with the beginning of the storyboard. The LTS provides a convenient mechanism for specifying synchronization among multiple media streams and for random access within a storyboard by using logical time as an address.

Each stream is composed of a sequence of *clips,* called a *cliplist.* For example, the video stream in figure 4-2 is composed of three clips, labeled A, B and C. A clip

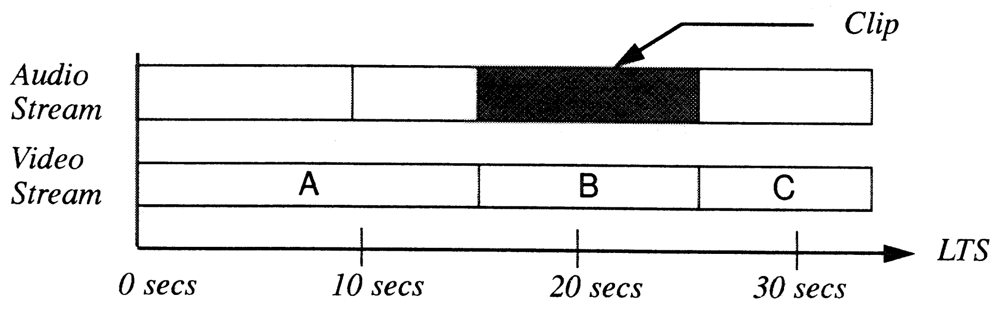


Figure 4-3: Sample Storyboard

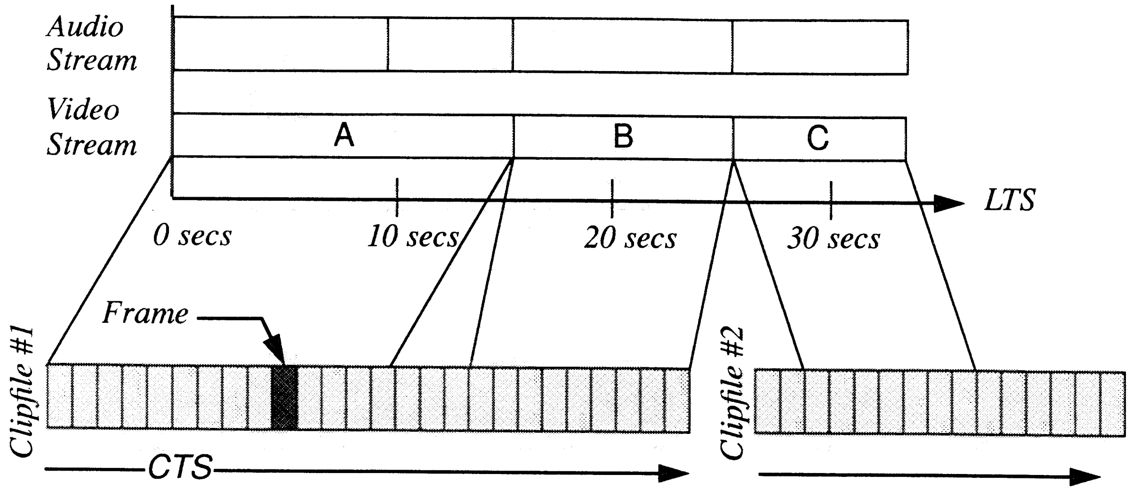


Figure 4-4: CMT Data Model Elements

is a segment of a *clipfile,* which is a file that stores a sequence of *frames* (a playable unit of data). For example, a video clipfile stores a sequence of images, and an audio clipfile stores a sequence of audio samples. Each clip is represented as a tuple of the form *<host, filename, start, end>*, where *host* is the host name of the file server where the clipfile is stored, *filename* is the full path name of the clipfile on that host, and *start* and *end* are the start time and end time, respectively, of the segment of interest. The start and end times are specified relative to the Clipfile Time System (CTS) which is the coordinate system for specifying time within a clipfile. The origin of the CTS (time=0) is the first frame in the clipfile.

Table 4-2 and figure 4-2 summarize these terms.

Clipfile Format

All CM data is stored in clipfiles. Clipfiles are designed to simplify random access, support efficient reading, and enable reuse of clips in storyboards. A sample clipfile is shown in figure 4-2[1]. It has five parts: a media-independent header, a me-

[1] The byte addresses shown in the figure are exemplary; the actual byte addresses vary among clipfiles.

Term	Definition
sample	The smallest atomic unit of data (e.g., a single 8 bit audio sample).
frame	The smallest playable unit (e.g., a group of audio samples or a video frame)
clipfile	A sequence of frames stored contiguously.
clip	A subsequence in a clipfile.
stream	A sequence of clips of the same media type.
cts	The clipfile time system, for specifying offsets into a clipfile, where t=0 is the beginning of the clip.
lts	The logical time system -- the horizontal axis in a storyboard.
storyboard	A series of streams synchronized along an lts.

Table 4-2: CMT Data Model Terms

dia-dependent header, raw data, the *frame offset table* (FOT), and the *frame time table* (FTT).

The media-independent header stores information common to all clipfiles. This information includes a magic number identifying the file as a clipfile, the type of media (e.g., video or audio), the format (e.g., video formats such as motion JPEG or MPEG, or audio formats such as Sun Audio files of MPEG audio), the number of frames in the clipfile, the clipfile version number, the address of the FOT and FTT in the file, and the number of frames within the clipfile. The media-dependent header contains media specific information such as image dimensions for video and sampling rate for audio. The format of this section and its content depends on the type of media stored in the clipfile. Table 4-3 lists examples of information stored in the media-dependent header for a few data types.

Raw data follows the header information. This section contains a contiguous sequence of binary data formatted in a media-dependent way. For example, the raw data in an MPEG clipfile contains a valid MPEG bitstream. The raw data section is partitioned into frames for efficient retrieval and transmission, since, in some

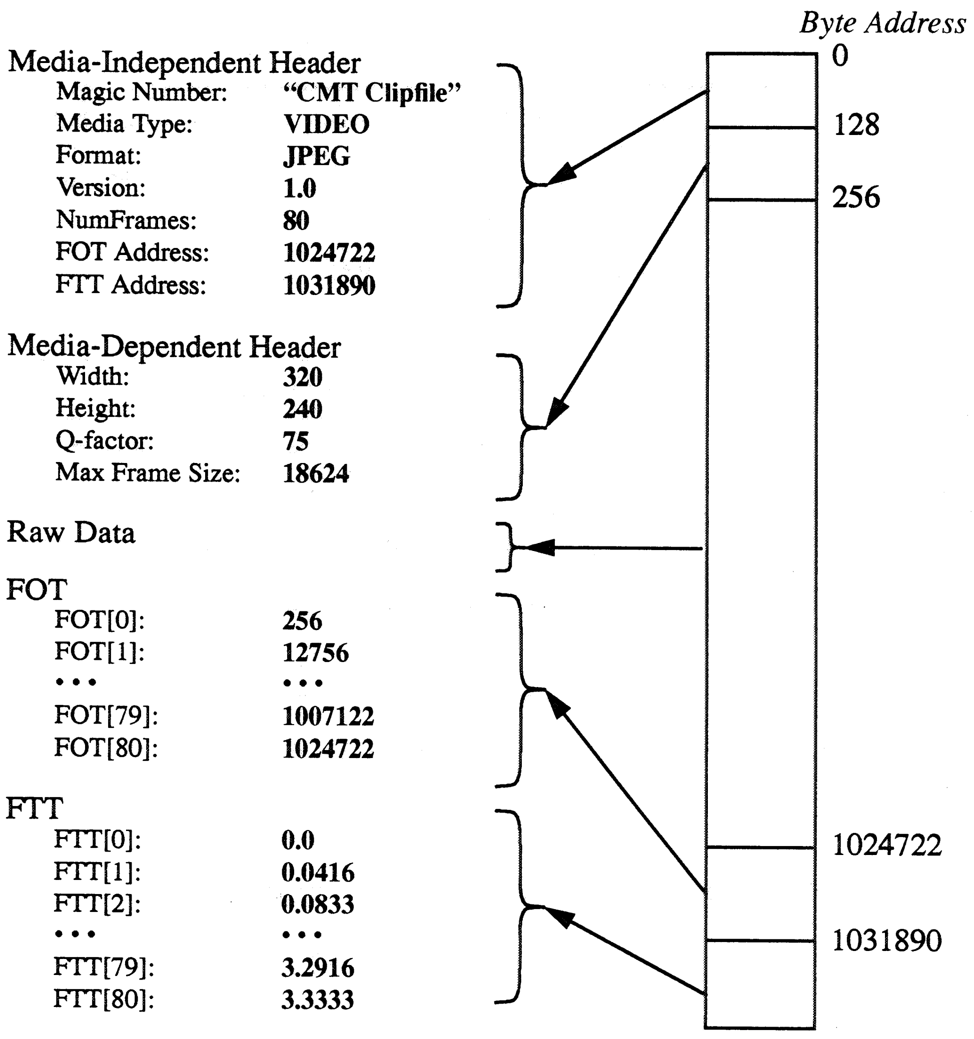


Figure 4-5: CMT Clipfile Format

Media	Data
Sun Audio	Sample rate, bits/sample, channels, optimal gain
Motion JPEG	Image width/height, quantization information, maximum frame size (bytes)
MPEG	• Image width/height, encoding pattern, • Sequence header table (bytes offset and size of all sequence headers) • Group of pictures table (bytes offset and size of all GOP headers) • Frame information table (type and reference frames for each MPEG frame)

Table 4-3: media-dependent header information

formats, a single sample is too small a unit for efficient transmission (e.g., 1-2 byte samples for audio). In such cases, larger groups of samples may be used as frames. The size of a frame is chosen to be large enough that transmission overhead is negligible.

The final section of the clipfile contains the FOT and FTT tables, which are indices used for random access into a clipfile. The FOT is a sequence of *N+1* integers, where *N* is the number of frames in the clipfile, that specify the address of the corresponding frame. The difference between two consecutive FOT entries gives the size of a frame. Thus, the last entry in the FOT is the byte offset of the FOT itself. For example, to read frame *m*, you would seek to FOT[*m*] and read (FOT[*m+1*]-FOT[*m*]) bytes.

The FTT is an array of *N+1 time values*, where *N* is the number of frames in the clipfile. FTT[*m*] is the start time of frame *m*, and FTT[*m+1*]-FTT[*m*] is the duration of frame *m*. The time values are relative to the CTS, so FTT[0] is always 0. Time values are stored as two 32 bit integers representing seconds and nanoseconds. This representation allows time values up to 2^32-1 seconds (about 136 years) to be represented to nanosecond accuracy. Together, the FOT and FTT provide indices to efficiently retrieve the frame bound to a given time from the clipfile.

Note that the FTT can be computed for CM data with constant audio and video display rates. In the case of animation data, however, the display rate is not constant, since some frames may be displayed many times longer than others. CM data captured by imperfect systems or from imperfect sources (e.g., data recorded from a bursty video conference) also requires an FTT, since the frame rate may be irregular. I therefore chose to put the FTT in all clipfiles to provide a uniform file format.

Comparison with other formats

The clipfile data structure encapsulates a contiguous sequence of frames of a single media type. Once a clipfile is created, it may be used in storyboards. The clipfile/storyboard structure has the following advantages over other data formats:

1. It facilitates editing.
2. It can save storage space.
3. It can save network bandwidth.

Editing operations fall into three categories: assembly, synchronization, and special effects. Assembly editing is the process of inserting or deleting a clip into a stream. Synchronization editing is the process of adjusting the relative timing of two streams. Special effects editing is the process of generating or modifying CM data, such as in scene transitions (e.g., fading a sequence), composition (e.g., chroma-keying), or simulation (e.g., ray tracing).

The clipfile/storyboard representation facilitates assembly and synchronization editing because the information needed to perform these edits is kept separate from the CM data. In other formats such as AVI from Microsoft [80,59], MPEG [44], and the JMovie structure from Parallax [56], synchronization and assembly information is maintained by interleaving data within a single file. To perform assembly or synchronization editing, a new file must be produced for each edit, which can be a time consuming process owing to the large size of video data. In contrast, assembly or synchronization editing using the storyboard/clipfile structure requires the production of a new storyboard, which is a straightforward task.

The storyboard representation makes assembly and synchronization of media recorded at different or irregular frame rates possible, because video segments are specified using precise time values, not frame numbers as in other systems (e.g., SMPTE). This feature allows fractional frames to be retrieved, and media recorded at different or irregular frame rates to be combined. Irregular frame rates can occur in video digitized by a non real-time capture system (e.g., desktop video capture systems or real-time video generation systems such as [27]) or in media such as animation or slide show that may have no regular period for frame update. Other formats specify a single frame rate for the entire video sequence, which forces resampling of video recorded at different or irregular rates. Depending on the encoding, resampling can be a expensive proposition.

The storyboard scheme can save storage space. In interleaved schemes, stock footage that appears in several videos (e.g., a studio logo) is stored multiple times, once for each video, which may consume large amounts of disk space. Different versions (e.g., different edits) of the same film are also stored multiple times. For example, consider an educational video archive with 1000 hours of 1 hour lectures. Suppose each lecture has 1 minute of stock footage at the beginning of each tape identifying the educational institution. Over the entire collection, this repeated footage amounts to a collective 1000 minutes of replicated information. Since one hour of MPEG compressed video typically requires about 1 GByte of storage, the replicated data consumes about 1000/60 = 17 GBytes. With storyboards and clipfiles, the stock footage can be stored once and shared among the lectures.

The storyboard scheme can save network bandwidth in some cases. For example, a multilingual video can be stored as multiple storyboards, one storyboard for each language with shared video clipfiles. When a user requests playback of the video in a particular language, only the audio and video data appropriate to the selected language is sent. Since storyboards take up little space (one hour of audio/video material broken into one minute clips typically uses less than 4 KBytes), storing multiple storyboards is practical. In interleaved formats, the multilingual problem is often solved by storing multiple copies for each language (an expensive alternative) or by storing multiple audio tracks in the file, one for each language, and playing a single audio track. Removing the extraneous audio tracks involves parsing the interleaved file, so most file servers send the full bitstream, including the unused audio tracks, to the client. Since the audio data in the unused tracks is dropped, network bandwidth is wasted.

But the most compelling reason for a representation similar to the storyboard representation is this: CM toolkits should be flexible in the file formats they support, because many media encodings are currently available, each offering a different trade-off of quality versus bitrate, and users will want to combine media with different encodings. A good CM toolkit should insulate the application programmer from the details of media encoding. If the toolkit does not provide the functionality to combine media types explicitly, application developers will develop their own, most likely incompatible, methods for providing this function.

The advantages of the storyboard representation comes at a cost. Storyboard representations are subject to referential integrity problems because a storyboard may be left with dangling pointer if one of the clipfiles it references is deleted. Furthermore, replicating a presentation may require copying many data files (the *copy* problem). The referential integrity problem can be solved by using a database manager to guarantee that referential integrity within a storyboard is maintained when the storyboard or its clipfiles are modified, moved, or deleted. The copy problem can be circumvented by a utility that assembles a new clipfile from the relevant portions of the component clipfiles specified by the script. The assembled clipfile can then be copied to the target system.

4.3 Implementation

This section describes the implementation of CMT and the CM Player. I first describe the changes made to the graphical user interface (GUI) toolkit on which CMT is based. Next, I describe the user visible and internal objects CMT provides to support the storyboard model, and then show how these objects are mapped to the three processes described in the introduction, namely the Application, CMX, and CMS, and how they interact with one another. Finally, I show how these components can be used to build a wide variety of playback-oriented applications.

Infrastructure

CMT is an extension to the Tcl/Tk graphical user interface (GUI) toolkit. Tcl (a "tool command language," pronounced "tickle") provides an embedded interpreter for programs written in the C programming language]. Tk (pronounced "tee kay") is Tcl's GUI toolkit for the X window system [36]. Together, Tcl and Tk provide a flexible environment for building GUIs. Common user interface components, such as buttons, menu bars, and the like, are provided by Tk as commands in the Tcl language. This section describes the extensions I made to the Tcl/Tk event processing model, the new event types I introduced, and the inter-process communication mechanism I developed for CMT.

Event Loop Modifications

Like many GUI toolkits, Tk is an event driven system, which means a callback routine can be called in response to events from the X server, file-events (i.e., when a socket or file becomes readable), timer-events (i.e., when the system clock reaches a specified time), and idle-events. Idle-events are not external events, but rather code to be evaluated when the system has no other useful work to do. The main loop of the program, called the *event loop,* waits for an event to be received, looks up a *callback* routine in a table using the event as a key, and invokes the callback routine. In Tcl/Tk, the callback is often a Tcl command to be evaluated.

Figure 4-2 shows a pseudo-code version of Tcl/Tk's event loop. The loop processes all file-events first, using a global variable named *fileSet* to keep track of the set of files that have input pending. When the input on a file is processed, the file is deleted from the fileSet. File-events include window system events (i.e., X-events) and other network events[2].

After processing all active files, the callback associated with any expired timer-event is invoked. When the file and timer-events are exhausted, idle-events are processed. The call to *ReadyFiles* polls the system for any pending file-events that may have arrived during timer-event processing. Only those idle-events currently in the queue are processed in this pass through the loop. In other words, the processing of any new idle-events enqueued during idle-event processing is delayed until the next pass through the loop. When all available idle-, X-, file- or timer-events have been processed, the event loop issues a *select* system call to wait for an event. The time-out for the call is based on the next timer-event.

Although the event loop works well for most GUI applications, several changes were needed to adapt the loop to CM applications. The first change made to Tk involved the introduction of a new type of timer-event, called an *at-event*. At-events are used for a variety of soft real time scheduling tasks, most notably to call the appropriate function to play a frame of CM data. At-events differ from timer-events in several ways. First, at-events specify that a callback should be invoked when the system clock reaches a particular time, whereas timer-events specify that a callback should be invoked after a specified delay period. Two side effects of this subtle difference are that creating an at-event is more efficient than creating a timer-event, and an at-event is more accurate in timing of the callback. Creating an at-event is more efficient than creating a timer-event because timer-event creation requires two more system calls to read the system clock than at-event creation (an

[2] X-events are treated as file-events because they arrive on a socket

```
while (TRUE) {
doFiles:
   while (fileSet not empty) {
      file = GetElement(fileSet);
      fileSet = fileSet - {file};
      InvokeCallback(file);
   }

   while (timerQueue not empty AND
          timerQueue.eventTime <= ReadSystemClock) {
      InvokeTimerCallback (Dequeue(timerQueue));
   }

   if (idleQueue not empty) {
      if (ReadyFiles() != NULL) {
         fileSet = ReadyFiles();
         goto doFiles;
      }
      lastIdleCallback = idleQueue.tail;
      repeat {
         callback = Dequeue(idleQueue);
         InvokeIdleCallback (callback);
      } until (callback == lastIdleCallback);
   }

   if (timerQueue != NULL) {
      timeOut = timerQueue.eventTime - ReadSystemClock();
   } else {
      timeOut = INFINITY;
   }
   select (fileSet, timeOut);
}
```

Figure 4-6: Main Event Loop in Tcl/Tk

important consideration if many at-events are created). At-events allow more accurate timing of the callback because timer-events must calculate the time of callback by reading the system clock and adding the specified delay to this value. The timer-event will fire late by an amount equal to the delay between the time the user issues the call to create the timer-event and when the system clock is read. This delay can be significant owing to context switches or page faults. Such unpredictable behavior will have an adverse affect on media synchronization.

A second difference between timer-events and at-events is that at-events specify a window of time within which the callback must be invoked and an optional callback if the system misses the window. The event window allows the system to shed load when scheduled events cannot be processed. The missed at-event callback allows the system to track missed events or take corrective action. Finally, at-events are run at a higher priority than other event types, improving synchronization.

One problem encountered in using the system was that all file-events were processed at a lower priority than at-events. This predetermined prioritization presented a dilemma of which events should be processed at higher priority: file-events or at-events? One argument says that at-events are higher priority than file-events, since at-events are timed constrained by nature and file-events are not. Furthermore, since the most common use for at-events is for synchronization, and the most common use for file-events is receiving the high bandwidth CM data from the file server, synchronization might suffer if the system is overloaded with data. But *some* file-events are higher priority than at-events, because control information sent from another process (such as the user pressing the *stop* button or network flow control data) is received as file-events.

The solution to this dilemma was to introduce named *priority groups.* A list of priority groups, called the *priority list,* is maintained by the toolkit. Applications can create new priority groups and set the order in the priority list. When a callback for an event is registered, it is put into a specified priority group.

In the event loop, callbacks are invoked in priority list order. Within a given priority group, events are processed in the original Tk order: at-events first, then file-events, timer-events and finally idle-events. All pending events within a priority group are processed before the next group in the priority list is examined. Any callbacks created while executing a callback are placed in the priority group of the executing callback.

Priority groups solve the problem of which events to process first by giving the programmer more flexibility. In the CMT, a high priority group is created for sockets associated with control information and a lower priority group is created for at-events associated with media playback and reception of CM data sent from the file server. Since control information is infrequently sent, and usually quickly processed, processing it at high priority has little effect on media synchronization.

Tcl-DP

The second change made to Tcl/Tk was the introduction of Tcl-DP. Tcl-DP is a distributed programming extension to the Tcl/Tk toolkit. Tcl-DP provides a Tcl interface to the Unix socket abstraction, an easy-to-use model for creating client/server applications, a remote procedure call (RPC) facility, a form of concurrent programming (RDO), and a simple distributed object system. Tcl-DP is the glue that links together the various processes within CMT.

RPCs in Tcl-DP take the form of Tcl commands that are evaluated by a remote interpreter. The return value (or error code) from the remote evaluation is returned to the caller. The communication protocol is based on TCP/IP and is therefore reliable. Tcl-DP supports time-outs on RPCs. If requested, a user specified callback routine is invoked should the RPC not return after a specified period of time. Since Tcl is an interpreted language, RPCs are implemented by passing string data between processes, which allows rapid prototyping and experimentation without the stub compilers needed in other RPC packages.

Tcl-DP provides a non-blocking form of RPC, called RDO (for "Remote DO"), that allows for concurrency in distributed processes. The semantics of RDO are the same as RPC, except no value is returned and the calling process does not wait for a reply. RDO invokes a co-routine on the remote machine. RDO can also be used to invoke a function on the server that returns a value. The client continues processing while the return value is being computed, and the return value is passed to a callback routine, which is a continuation for the co-routine.

The performance of the RPC implementation is competitive with other RPC implementations. Current measurements on a Sun Sparcstation show that typical RPC calls add about 4 milliseconds to the overhead of a procedure call. RDOs, in contrast, require about 500 microseconds to complete. RDOs are faster than RPCs because they are non-blocking.

Distributed CM applications are composed of one or more communicating processes. Although the Tcl-DP RPC mechanism provides a communication method for these processes, I found it more convenient to hide much of the communication details by using a distributed object management system (DOMS) built using Tcl-DP. The Tcl-DP DOMS provides a mechanism to update object slots and to attach triggers to the slot updates.

The Tcl-DP DOMS takes a minimalist approach. It does not provide inheritance, persistence, automatic migration, name lookup, or automatic replication in the event of crashes; it includes only those features needed to support the CMT. Although not particularly interesting as a DOMS in its own right, the Tcl-DP DOMS is interesting because it shows how much can be done with a simple DOMS and what DOMS features are needed to implement CMT.

In the Tcl-DP DOMS, an *object* is a named collection of slots that follows a simple slot access protocol. Objects are instantiated using a command that creates an object with a specified name. The name of the object becomes a new command within the interpreter. Slots are read using the *getf* command followed by the name of the object and the slot of interest, and written using the *setf* command, which takes an object name, a slot name, and the slot's new value as parameters.

Once created, an object can be replicated in other processes connected by Tcl-

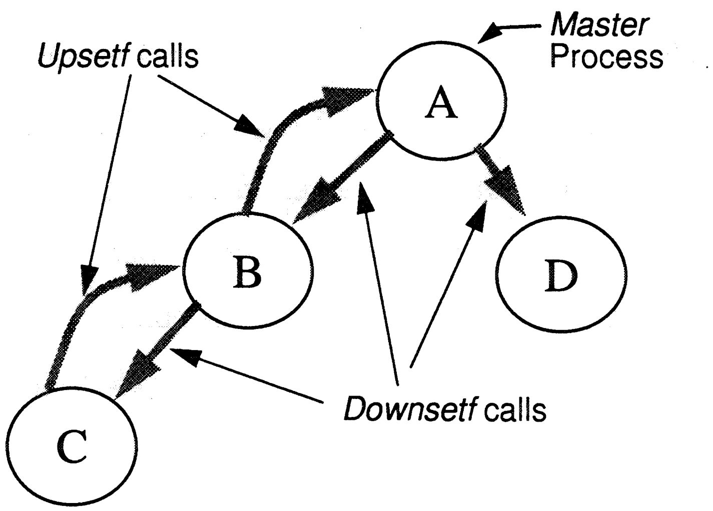


Figure 4-7: Distributed Object Updates in Tcl-DP

DP. Slot updates are serialized through a *master* process, which is the process that originally created the object. Each process in which the object exists uses a global table, indexed by object name, to associate a single *owner* process and zero or more *client* processes with each object. The owner process is the process from which the object was distributed, and the client processes are the processes to which the object was distributed.

Figure 4-2 shows the communication during an update of an object distributed to four processes. In this figure, A is the master process. If a slot is updated by process C, the update request, called an *upsetf*, is passed up the distribution tree to each owner process, without being executed, until the master process (A) is reached. When A receives the upsetf request, it sends a *downsetf* to the client processes B and D, and the update is executed. Each of these processes passes the *downsetf* down the tree to their clients. In this example, B passes the update request to C, and C executes the update. Although object copies may be inconsistent for a short period during an update, the final state of each object is guaranteed to be consistent using this protocol.

A list of triggers can be associated with a slot of an object. Each trigger specifies when the trigger should fire (i.e., before or after the slot update), the name of the object, the slot of interest, and Tcl code to evaluate when the slot is updated.

One problem that came up while using distributed objects is name conflicts. For example, consider an object named *mybox* created in process A. Suppose A distributes mybox to process C. Now suppose another process, say B, creates another object named *mybox* and tries to distribute it to process C, resulting in two objects in C named mybox. This situation is called a name conflict.

The solution to this problem is to provide a method that allows the application to generate unique names for objects it wants to distribute. Each name has the form *<string><pid>.<id>@<host>*, where *<string>* is a human readable string, *<host>* is the internet address of the machine on which the object is created, *<pid>* is the process identifier of the process in which the object is created, and *<id>* is an incremental counter variable within that process. For example, the name of the object *mybox* created by process 2436 on toe.cs.berkeley.edu might be *mybox2436.2@128.32.149.117.* This pid/hostname mechanism is a simple, portable way of generating unique names for objects in distributed processes.

The Tcl-DP DOMS was used to implement CMT. Because it was built in response to the needs of CMT, it is lean; it contains few features not needed by the toolkit. Furthermore, because features were only added to the Tcl-DP DOMS when they were needed by CMT, the Tcl-DP DOMS represents the minimal DOMS features required to build a CM toolkit with an architecture similar to CMT. Tcl-DP is surprisingly small compared to other, full-featured DOMS systems. Table 4-4 lists the lines of C and Tcl code used in the current implementation of the Tcl-DP DOMS (version 3.1).

Module	Language	Lines
Basic Communication Library	C	3000
	Tcl	500
RPC Library	C	1700
	Tcl	300
Object System	C	0
	Tcl	500
Total	C	4700
	Tcl	1300

Table 4-4: Lines of code in Tcl-DP and the Tcl-DP DOMS

CMT Objects

CMT applications are composed of communicating objects that support the storyboard abstraction described in section 4.2. The objects include an object to represent the LTS of a storyboard, objects to represent the streams in a storyboard, and objects to implement the producer/consumer model for CM data used in CMT. This section describes the properties of each class of objects in turn.

CMT implements the storyboard abstraction using two objects: the *logical time system* (LTS) object and one or more *stream* objects. Each stream object has a cliplist and a pointer to an associated LTS object, as shown in figure 4-8. The cliplist is a list of clips, each of which specifies the location and portion of a clipfile. Both types of objects are distributed objects shared by the Application, CMX and CMS process.

An LTS has two slots: *speed* and *offset.* These slots specify a linear mapping from time on the system clock (*system time*, or *ST*) to time on the storyboard (*logical time*, or *LT*). The result of this mapping is called the *value* of the LTS, and represents the current position in logical time on the storyboard.

CMT uses the value of the to LTS locate the clipfile which corresponds to the current value of the LTS, called the *active* clipfile, and to decide what frames of the active clipfile should be displayed. CMT also uses the LTS to schedule the delivery and playback of the audio and video data, which is implemented by five objects associated with each stream: the *MediaSource*, the *PacketSource*, the *PacketDest*, the *MediaDest*, and the *MediaResource*. MediaSource and PacketSource objects are located in CMS, and PacketDest, MediaDest, and MediaResource objects are located in CMX, as shown in figure 4-8. The gray lines in the figure indicate the flow of CM data between these objects.

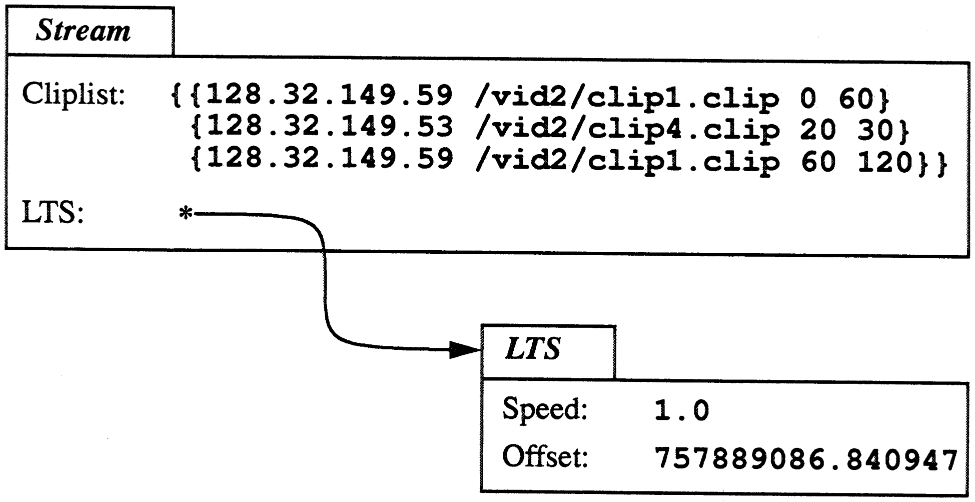

Stream
Cliplist:
{{128.32.149.59 /vid2/clip1.clip 0 60}
{128.32.149.53 /vid2/clip4.clip 20 30}
{128.32.149.59 /vid2/clip1.clip 60 120}}
LTS:
*
LTS
Speed: 1.0
Offset: 757889086.840947

Figure 4-8: CMT Stream and Cliplist Objects

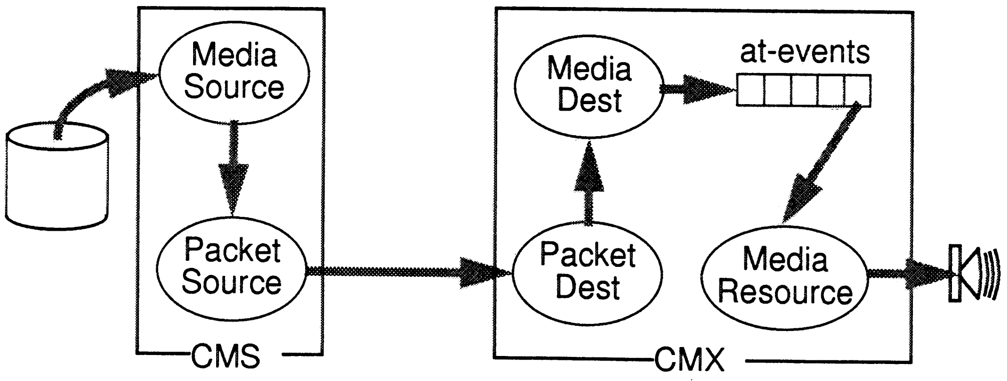


Figure 4-9: Media- and Packet- Sources, Dests, and Resources

Each of these objects serve a specific function in the playback of a frame of CM data. The MediaSource reads CM data from a local disk and passes the resulting frames to the PacketSource. Disk reads are scheduled using the LTS and dynamic network performance measurements. The PacketSource fragments frames and delivers them to the PacketDest using a custom protocol described in the next chapter. The PacketDest reassembles the fragmented frames and passes the reassembled frames to the MediaDest, which schedules decoding and playback using an at-event. This at-event calls a function in the MediaResource to output the frame.

This section is composed of three subsections. The first subsection describes the LTS object, the second subsection describes the stream object, and the third subsection describes the MediaSource, PacketSource, PacketDest, MediaDest, and MediaResource objects.

The Logical Time System

The logical time system (LTS) object represents the storyboard's LTS. An LTS has two slots: *speed* and *offset*. These slots specify a linear mapping from time on the system clock (*system time*, or *ST*) to time on the storyboard (*logical time*, or *LT*). The result of this mapping is called the *value* of the LTS, given by:

$$value = speed \times ST + offset \qquad \text{EQ 4-1}$$

The *value* of an LTS is the current position on the storyboard in logical time. Speed controls the rate at which logical time advances: 1) when speed equals zero, logical time is stopped, 2) when speed equals one, logical time advances at real-time rate, and 3) when speed is negative, logical time runs backwards.

The user can set or read the *speed* and *value* slots. The latter slot is a *virtual slot:* it requires no storage space, but can be read and written like a normal slot.

When the user reads the value slot, the result of equation 4-1 is returned. When the user writes the value slot, the offset slot is adjusted so the result of equation 4-1 matches the specified value. From equation 4-1, this means that

$$offset = value - speed \times ST \qquad \text{EQ 4-2}$$

When the speed of the LTS is changed, the continuity of logical time is preserved. That is, the value of LT after the change is the same as the value of LT before the change. For example, suppose the system is stopped at LT=5, and the value of ST is 1,000[3]. If speed is set to 1, offset must be changed to -995 to preserve the continuity of logical time. This constraint is expressed in the equation

$$speed' \times ST + offset' = speed \times ST + offset \qquad \text{EQ 4-3}$$

where *speed'* is the new speed, *speed* is the old speed, and *offset* is the old offset. The new offset, *offset'*, which preserves the continuity of logical time, is given by

$$offset' = ST \times (speed - speed') + offset \qquad \text{EQ 4-4}$$

This constraint is implemented as part of the assignment function for the speed slot of the LTS object.

The LTS allows basic VCR controls to be easily implemented. Normal playback corresponds to setting the speed slot to one. Fast forward, at various rates, corresponds to LTS speeds greater than one, stop corresponds to an LTS speed of zero, and reverse play, at various rates, corresponds to negative LTS speeds. Random access is accomplished by setting the value of the LTS, and single stepping is implemented by setting the LTS speed to zero, and incrementing or decrementing the value of the LTS by the duration of a single frame (e.g., one-thirtieth of a second).

[3] The value of ST in a Unix system is measured in microseconds since Jan 1, 1970.

The LTS also implements the inverse function of equation 4-1, which maps from logical time to system time. This function is called the *SystemTimeOf* function. When the speed of the LTS is non-zero, *SystemTimeOf* returns the value computed by the equation

$$(LT - offset) / (speed) \qquad \text{EQ 4-5}$$

Three special values are returned by *SystemTimeOf* when speed equals zero, since this equation is not defined in this case. The three values are:

$$\begin{array}{rl} infinity & LT < offset \\ -infinity & LT > offset \\ now & LT \equiv offset \end{array} \qquad \text{EQ 4-6}$$

Tables 4-5 and 4-6 summarize the slots and methods provided by the LTS object.

Slot	Description
Speed	Controls advance of logical time relative to real time
Offset	Internal slot used in linear mapping of LTS. Never directly set or accessed by user.

Table 4-5: LTS Slots

Method	Description
Value	Virtual slot that returns the current value of logical time per equation 4-1. Setting this slots adjusts offset per equation 4-2
SystemTimeOf	Returns the system time that corresponds to a give logical time.
LogicalTimeOf	Return the logical time that corresponds to a given system time.

Table 4-6: LTS Methods

Streams

A storyboard contains two types of objects: LTS objects and *stream* objects. Each stream object represents a CM data stream in the storyboard. A stream object has two slots: *LTS* and *cliplist.* The *LTS* slot stores the name of the LTS object that controls the stream. The *cliplist* slot contains a list of clips, each of which is either a *data* clip or a *blank* clip. A blank clip has the form *(blank, duration)*, where *blank* is a keyword identifying the clip as a blank clip and *duration* specifies the length of the blank clip, in logical time. Blank clips are used to specify a period in the stream when no CM data is output. A data clip is specified as a tuple of the form *(host, filename, start, end). Host* stores the internet address of the machine where the clipfile is stored, *filename* stores the full path name of the clipfile, and *start* and *end* specify the start and end times of the segment of the clipfile relative to the CTS.

Table 4-7 summarizes the slots provided by the stream object. The stream object provide no methods because the main use of the stream object is specify the cliplist to CMX. Stream objects in CMT are usually Tcl-DP distributed objects that are shared between the Application and CMX. When the Application sets the *cliplist* slot of a shared stream object, triggers in CMX detect this change and take appropriate action to display the selected data. This use of slots and triggers to provide communication between the two processes, in the case the Application and CMX, is called *implicit control* and is discussed further below.

Slot	Description
LTS	Specifies the LTS to which the stream is synchronized
Cliplist	Specifies a list of clips. Each clip specifies a segment of a clipfile.

Table 4-7: Stream object summary

Normally, one stream object is created for each stream in the storyboard. Although the *LTS* slot of these stream objects usually point to a common LTS object, this is not necessary. By using multiple LTSs, interesting applications can be implemented. For example, suppose that you wanted to synchronize a video stream with an audio stream, and to do so the video stream must play at a rate 10% faster than normal. Although you could create a new clipfile with the video resampled for this purpose, a less costly solution is to use two LTS objects, one associated with the video stream (the *video LTS*), and one associated with the audio stream (the *audio LTS*). The desired synchronization is then accomplished by setting the speed of the video LTS to 1.1 times the speed of the audio LTS. Other uses of multiple stream and LTS objects are discussed at the end of this section.

Media Sources/Dests/Resources and Packet Sources/Dests

CMT uses a producer/consumer model for playing CM streams. CMS is the producer; CMX is the consumer. The producer/consumer model is implemented in CMT by five objects: the MediaSource, the MediaDest, the PacketSource, the PacketDest, and the MediaResource.

The frame is read from secondary storage by a MediaSource object. The MediaSource understands the clipfile format and the media's properties. It uses this knowledge to decide what frames to read, and what frames to skip should insufficient bandwidth exist between CMS and CMX.

The MediaSource has three slots, a *cliplist* slot that contains the list of clips that the MediaSource will deliver to a client, an *lts* slot that stores the name of the LTS object and associated with the cliplist, and various structures to represent the network connection between CMS and CMX. The network structures are described in the next chapter. The *cliplist* slot specifies the portion of the CM data stream that is served by the CMS where the MediaSource is instantiated. It is computed from a *stream* object shared between the Application and CMX by creating one cliplist for each host specified in a stream's cliplist and extracting the set of clips served by that host. Blank clips are inserted where clips are served by other CMSs.

For example, suppose the cliplist of a stream object shared between the Application and CMX is

```
{{linus.cs.berkeley.edu /video/clip1.clip 0 60}
{gumby.cs.berkeley.edu /video/clip4.clip 20 30}
{linus.cs.berkeley.edu /video/clip1.clip 60 120}}
```

Two hosts are specified in this cliplist, the linus.cs.berkeley.edu and gumby.cs.berkeley.edu[4]. The cliplist of the MediaSource object on linus.cs.berkeley.edu would be

```
{{linus.cs.berkeley.edu /video/clip1.clip 0 60}
{blank 10}
{linus.cs.berkeley.edu /video/clip1.clip 60 120}}
```

and the cliplist of the MediaSource object on gumby.cs.berkeley.edu would be

```
{{blank 60}
{gumby.cs.berkeley.edu /video/clip4.clip 20 30}}
```

The MediaSource has four methods: *ItsChanged*, *create*, *destroy*, and *configure*. *Create* is used to instantiate a MediaSource; *destroy* deletes a previously created instance. *Configure* is used to change the set or read the values of the MediaSource's slots. The *ItsChanged* method is called when the either the *speed* or *offset* of the LTS object stored in the MediaSource is changed. These slots and methods are summarized in tables 4-8 and 4-9.

The MediaSource reads frames from a local disk a short time (typically 0.5 seconds) before they are needed for playback. Each frame is passed to the PacketSource, which fragments the frame and queues it for delivery. The PacketSource

[4] The internet address for linus and gumby would normally be specified, but I am using the host names for clarity.

Slots	Description
Lts	Stores the name of the LTS that controls the stream
Cliplist	The cliplist specifying the portion of a stream served by this MediaSource.
Network	Various structures representing the connection between CMS and CMX. Detailed in the next chapter.

Table 4-8: MediaSource Slots

Methods	Description
LtsChanged	Invoked when either the *speed* or *offset* of the controlling LTS are changed.
Create	Creates a MediaSource
Destroy	Destroys a MediaSource
Configure	Reader/writer method for the slots

Table 4-9: MediaSource Methods

sends the fragments to the PacketDest via a computer network, and the PacketDest reassembles the frames and requests lost packets. When the frame is complete, it is passed to the MediaDest for playback. Frame fragmentation and delivery are discussed in detail in the next chapter. For the purposes of this discussion, the only relevant property of the protocol used by the PacketSource and PacketDest is that it is unreliable. That is, it may not be possible to deliver a frame in time for playback.

When the PacketDest passes a completely reassembled frame to the MediaDest, the MediaDest performs any time consuming preprocessing needed for playback and schedules the frame for playback. An example of such preprocessing is software decompression, since waiting until playback time to decompress in software may result in synchronization problems. Since such preprocessing is media specific (i.e., it varies with the type and format of the CM data), a different Media-

Dest is required for each media type. The MediaDest can also intelligently resolve scheduling conflicts should insufficient resources exist to play all the CM data that arrives, and send feedback to CMS indicating that it should send less data (i.e., it decreases the bandwidth estimate between CMS and CMX). Details of this type of processing are presented elsewhere [67].

Scheduling of frames for playback is performed by converting the logical start and end times of the frame to system time using the LTS associated with the frame's stream, and a high priority at-event is created to play the frame.

When the time for frame playback arrives, the at-event scheduled by the MediaDest calls the MediaResource *play* method to play the frame. *Play* correctly resets the state of any decompression hardware to the corresponding MediaDest and plays the frame. The MediaResource thus multiplexes the underlying hardware, including the CPU for software decompression, between several MediaDests. Its internal state is highly media dependent. For example, the JPEG MediaDest contains software decompression tables. The *play* method should be engineered to implement frame display as promptly as possible. For example, in the JPEG MediaDest, this function simply calls the underlying hardware to decompress and display the frame, using shared memory with the X server to transmit the data. It then forces a context switch by waiting for an event indicating that the display operation is complete.

The architecture allows for a single stream to be served by several CMSs, since the MediaDest is blind as to where the frame given by the PacketDest originated. Thus, video can be striped across several file servers, making parallel reading and load balancing possible and simplifying cache management of video loaded from tertiary storage.

The architecture also allows new media types to be easily incorporated by creating new types of MediaSource, MediaDest and MediaResource objects. Since these objects are specialized to serve a specific type of media (i.e., they are *media-specific* objects), they can use this knowledge to selectively drop frames should insufficient bandwidth exist.

Finally, to port the system to different compression hardware, only the *play* method in the MediaResource must be modified to access the new hardware. Often, such a port involves changing only a few lines of code.

Implicit Control Using Distributed Objects

Although the Application, CMX and CMS can communicate directly using Tcl-DP, it is often more convenient to use an implicit communication mechanism that uses Tcl-DP DOMS distributed objects. This section describes how distributed LTS and stream objects are used to control CMS and CMX.

The clocks of all three processes are assumed to be synchronized. The current implementation uses NTP [49] for synchronization, but any synchronization protocol that keeps the clocks synchronized to within about 100 milliseconds would suffice.

After creating the necessary LTS and stream objects, the Application distributes them to CMX using the Tcl-DP DOMS described above. The configuration of the CM Player after object distribution is shown in figure 4-10. The objects are created using the pid/hostname mechanism described in above to prevent name conflicts. When CMX receives a stream object, it attaches a callback to the cliplist slot of the stream object. When the Application modifies the cliplist slot of the stream, the change is propagated to CMX, triggering the callback. The callback causes CMX to tear down unnecessary connections with CMS processes and establish new ones. CMX passes a modified cliplist to each CMS. The modified cliplist has blank clips inserted at points where the stream is served by some other CMS. The LTS

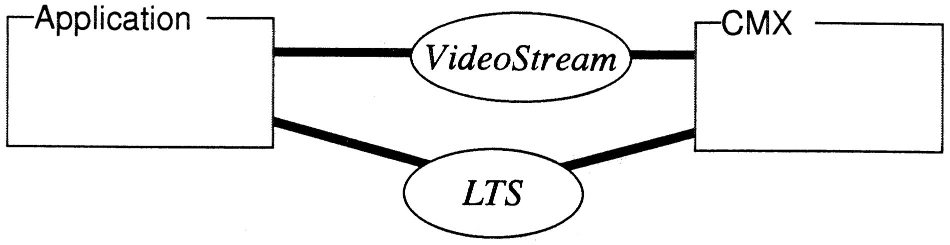


Figure 4-10: Shared Objects at CM Player Start-up

associated with the cliplist is also distributed to each CMS.

For example, suppose the Application creates an LTS object named lts2243.1@128.32.149.117 and a motion JPEG video stream object named jpeg-Stream.2243.2@128.32.149.117. For brevity, we will use the names *VideoStream* and *Lts* for these objects. *VideoStream* initially has an empty cliplist and the LTS slot points to *Lts*. Both *VideoStream* and *Lts* are distributed to CMX.

Now, suppose the Application sets the cliplist slot of *VideoStream* to

```
{{linus.cs.berkeley.edu /video/clip1.clip 0 60}
{gumby.cs.berkeley.edu /video/clip4.clip 20 30}
{linus.cs.berkeley.edu /video/clip1.clip 60 120}}
```

This change is propagated by the Tcl-DP DOMS to CMX, which triggers a call-back that establishes connections to the CMSs on linus and gumby. CMX passes the modified cliplist

```
{{linus.cs.berkeley.edu /video/clip1.clip 0 60}
{blank 10}
{linus.cs.berkeley.edu /video/clip1.clip 60 120}}
```

to linus and

```
{{blank 60}
{gumby.cs.berkeley.edu /video/clip4.clip 20 30}}
```

to gumby. *Lts* is also distributed to the CMS on linus and gumby. The configuration after this exchange is shown in figure 4-11.

Notice the style of interaction between the Application and CMX. Inter-process

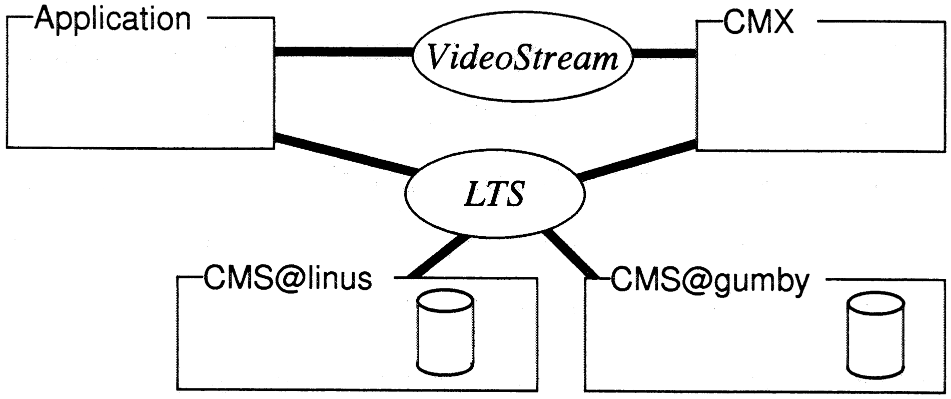


Figure 4-11: Shared Objects after setting cliplist (see text)

communication is done implicitly through the Tcl-DP DOMS, not explicitly via RPCs. The Application makes no explicit request to CMX to connect to a CMS. This style of interaction provides a powerful layer of insulation between the Application and CMX. CMX can change its internal implementation (if this is desirable) and no changes are needed in the Application. In essence, the distributed object is the API. The same style of interaction is used throughout CMT, and it is an important idea in the system. For example, to change the volume of the audio output, the Application sets the volume of the AudioStream. When this change is propagated to CMX, it triggers a callback that calls the appropriate method in the audio MediaResource to implement the user's request.

Similarly, the system uses the LTS, in combination with synchronized clocks (e.g., synchronized using NTP), to communicate user time control requests, schedule delivery, and maintain inter-media synchronization. CMS attaches a callback to the speed and offset slots of the LTS. When the Application changes these slots (for example, by pressing the *play* button), the change is propagated to CMX and from there to each CMS that has a copy of the LTS. A trigger calls the MediaSource's *ltsChanged* method which examines the LTS, calculates the time it should read the first frame it should to the CMS, and creates an at-event to read the frame at this time.

Returning to the example depicted in figure 4-11, when linus gets the message *set speed = 1*, the MediaSource *ItsChanged* method examines the cliplist and notices that it should begin sending data immediately, which it does. Meanwhile, the callback in the MediaSource on gumby examines its cliplist and creates an at-event to read the first frame in about 60 seconds.

The interaction of the Tcl-DP DOMS and the objects described above has proven to be a powerful method for building playback-oriented applications. Even though the only application we have currently implemented is the CM Player, the CMT architecture can be used to implement other playback oriented applications, as shown in the next section.

Other Applications

The Stream and LTS objects can be used to build other applications. For example, consider a simple assembly editing application that presents the user with two storyboards and allows sections of one to be pasted into the other. Here, the Application would create two LTSs, one for each storyboard, two JpegStreams, and two AudioStreams. The position within each storyboard can be set independently using the two LTSs, and editing is implemented as cliplist manipulations. For example, inserting a video clip from a *source* JPEG stream into a *target* JPEG stream is a three step process. First, the cliplist representing the section of the source stream you want to insert is constructed. Second, this cliplist is inserted into the appropriate position in the target cliplist. Third, the cliplist of the target Stream objects are set to their new values. The system takes care of the rest. Simple synchronization editing can be performed by modifying cliplists.

Another application that can be easily created using is a multilingual playback application. This application can be implemented in 2 ways. In one implementation, two LTSs are created: the *zero* LTS (it will always have speed equal zero) and the *playback* LTS. A single JpegStream is created referencing the playback LTS, and multiple AudioStream are created, with all inactive audio streams referencing the zero LTS and the active audio stream referencing the playback LTS. Control is the same as in the CM Player Application, but only the playback LTS is changed. To switch languages, the LTS of the current AudioStream is set to the zero LTS, and the LTS of the desired AudioStream is set to the playback LTS. This application can also be implemented by using a single LTS and changing the cliplist of the AudioStream.

A third application can perform synchronous playback of streams of different durations. For example, suppose you wanted to play a certain video clip synchronized with an audio clip, but the video clip is twice as long as the audio clip. Such synchronization can be implemented by creating two streams, a VideoStream and an AudioStream, each with a private LTS. Playback is performed by setting the speed of the VideoStream's LTS to twice that of the AudioStream's LTS.

Using multiple storyboards, hypermedia systems can be implemented. Each node in the hypermedia web would correspond to a storyboard. Visiting a node is then similar to selecting a movie in the CM Player. By using multiple LTSs and storyboards, complex hypermedia applications with non-linear playback schedules can be supported. For example, suppose you wanted to implement a schedule that played video segment A, then displayed a dialog with two buttons, waited for user input, and then played either storyboard B or C depending on which button the user presses. Storyboards B and C have both audio and video streams, and the audio segment in B must be played ten percent faster than the corresponding video segment. This schedule can be represented using the directed graph shown in figure 4-12. Switching storyboards in the dialog is analogous to visiting nodes in a hypermedia web, and the method for synchronizing streams with different playback rates using multiple LTSs was discussed in the previous paragraph. Such graphs are generated by multimedia compilers such as FireFly [11,12].

4.4 Related Work

Other research groups have attempted to develop toolkits for distributed multimedia applications. The literature is replete with requirements papers, language definitions, object-oriented extensions, data flow models, and sketchy design documents for toolkits, but few designs have been implemented. Steinmetz reviews some of the models [74].

For example, Bates and Bacon [6,7] have developed a language for describing a distributed multimedia presentation in terms of named pipelines that specify the connectivity of objects. The language also allows the binding of user-defined events in the CM data stream (e.g., "frame 52 plays") with actions. The runtime environment, called IMP, interprets the specification, instantiating the objects of the pipeline and creating additional entities that "watch" the CM data stream for the user-defined events. The language can be used to create multimedia, hypermedia, and computer supported cooperative work applications.

Other systems are closer in goals to CMT. The Amsterdan Multimedia Frame-

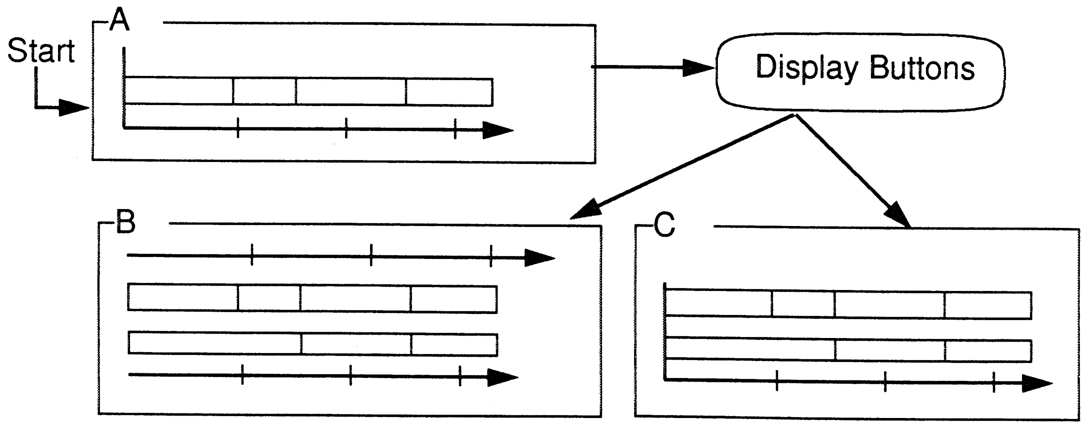


Figure 4-12: Hypermedia web with storyboard nodes

work (AMF) is one such system [13, 65, 30]. AMF defines a data model that provides a method for collecting data objects into multimedia documents called CMIFs. The CMIF format provides two views of multimedia data, the hierarchical view and the channel view. The channel view is analogous to the storyboard representation, except that CM streams are synchronized explicitly using synchronization arcs. A synchronization arc specifies a timing relationship between points in two multimedia streams. For example, a synchronization arc might specify that a certain video frame should be played at the same time as a given audio sample.

AMF also contains a system to play CMIF documents. This system consists of intelligent information objects (IIOs), local operating systems (LOSs), a global operating system (GOS), and applications. IIOs are producers, and applications are consumers. IIOs are containers for multiple representations of a single, logical piece of information. They provide a standard interface that an application can use to access a preferred representation. The GOS and LOSs allocate shared resources such as network bandwidth and buffer space among competing applications and IIOs. The LOS is analogous to the CMX and CMS processes in CMT. CMT has no analog of the GOS, since each application competes for the primary shared resource, network bandwidth, on a best-effort basis.

The Interactive Multimedia Association's (IMA) Multimedia System Services (MSS) proposal [51], defines a standard way to create and to control complex information flow graphs where nodes in the graph represent sources, sinks, and processors of multimedia data and edges in the graph represent CM connections between the nodes. The MSS is based on the Common Object Request Broker Architecture (CORBA) [84,50]. It defines a standard way of locating, creating, connecting, and controlling distributed objects that represent a wide variety of live and playback devices. The classes of objects include Virtual Devices, Virtual Connections, Virtual Clocks, Groups, Streams, Formats, Factories, and a Registration and Retrieval (R&R) Service. Virtual Devices are analogous to Sinks and Sources in CMT, but include support for live devices and multiple data formats within a single virtual device. Virtual Connections are similar to PacketSource/Dest pairs in CMT in that they connect data sources to sinks, but Virtual Connections provide high-level flow control semantics like pause and resume. The actual transport mechanism is unspecified, but a mechanism for communicating various quality of service (QOS) parameters is provided. A Stream object provides an access point for inquiry and control of a media stream. Its function is similar to CMT's LTS object. Virtual Clocks are used to synchronize media items in a stream, Group objects are used to control a group of objects as a whole, and Format objects provide an interface to query the specific media format (e.g., an image's width and height) a Virtual Device is using. Finally, Factories and R&Rs are used to locate and instantiate these objects.

The IMA's contribution is largely in standardization, since they ignore many important systems integration issues. For example, the next chapter will show that close coupling of file and transport services can lead to high quality playback. Such close interaction is difficult to implement in the IMA proposal since the file and transport objects are independent. The MSS has the potential to be an important standard in future multimedia systems but implementations and products are not being aggressively pursued by industry, despite the existence of prototypes.

The Heidelberg multimedia project at the IBM European Networking Center [19,32,33] contains a toolkit as part of its suite of tools for constructing CM applications. The HeiProject software is not designed to be portable: it uses a modified operating system to guarantee performance to applications. Like CMT, it is composed of sets of communicating objects. Classes of objects provide the following functions:

1. HeiSMS is their stream management system. It manages CM

data flow -- its function is similar to CMT's MediaSource, MediaDest, PacketSource, and PacketDest modules.

2. HeiBMS is their buffer management system. It handles most memory management functions and contains functions to minimize the copying of data. In CMT, individual objects manage their own memory, passing pointers to avoid data copying. This ad hoc solution presented problems, and a more recent version of CMT contains an explicit buffer manager module.

3. HeiRMS is their resource management system. It allows resource reservation and prevents over-booking resources. MediaResources serve this function in CMT.

4. HeiOSS is their operating system shield. It is primarily a library of common functions.

5. HeiTS and HeiTP is their transport system and transport protocol, based on ST-II. A comparison of CMT's transport layer with HeiTS is given in section 5.5 of the next chapter.

6. HeiToolkit is their toolkit for constructing distributed applications. It is similar in function to Tcl-DP.

Overall, these architectures are similar to CMT: distributed objects are connected in a pipeline along which CM data flows. The primary contributions of this work are: 1) CMT is portable -- it runs on unmodified operating systems and networks, 2) CMT uses objects for distributed control in a novel way, 3) CMT shows how to retrofit existing GUI toolkits for CM applications, and 4) CMT provides a data model that simplifies the editing, reuse, and sharing of data.

Chapter 5
Network protocols in CMT

5.1 Introduction

This chapter describes the network protocol CMT uses to deliver continuous media (CM) data in applications like the CM Player. The goal of the protocol is to provide high quality playback over an unreliable network without modifying the operating system or network router software. Protocols that provide this service are called *best effort* protocols.

One might wonder why I developed a special protocol for CM data delivery. Indeed, some research systems [31,20] use TCP/IP [75] for communication, including early versions of CMT. Experience with this implementation revealed that CM applications based on TCP are unstable with respect to the real-time bandwidth required by a media stream. When the required bandwidth is well below the available bandwidth, TCP connections perform well. When the required bandwidth is above the available bandwidth, they perform miserably. And, when the required bandwidth is about the same as the available bandwidth, performance is extremely sensitive to transient network loads. The instability is caused by the stream semantics of TCP which delays delivery of subsequent data if intermediate data is lost. So when the network is overloaded by a CM application, CM data is delayed to the point where frames arrive so late as to be useless.

Other researchers are developing real-time protocols to solve this problem [22], but they require changes in OS kernels and network routers and gateways that are not yet available. Hence, I developed a best-effort protocol for CMT that utilizes the existing IP infrastructure.

The CMT protocol differs from other best effort protocols because it is designed to support media-specific network protocols, which are protocols that use properties of the media and its encoding to achieve higher quality playback by compensating for the unreliable delivery channel. A simple example will illustrate the advantages of media-specific network protocols.

The MPEG video standard [44] uses differential coding between frames to achieve high compression ratios. MPEG defines three types of video frames: I-frames, P- frames, and B-frames. I-frames can be decoded independently, whereas P-frames require a previous I- or P-frame (called a *reference* frame) for decoding, and B-frames require two reference frames for decoding, one from the past and one from the future. These dependencies are illustrated in figure 5-1.

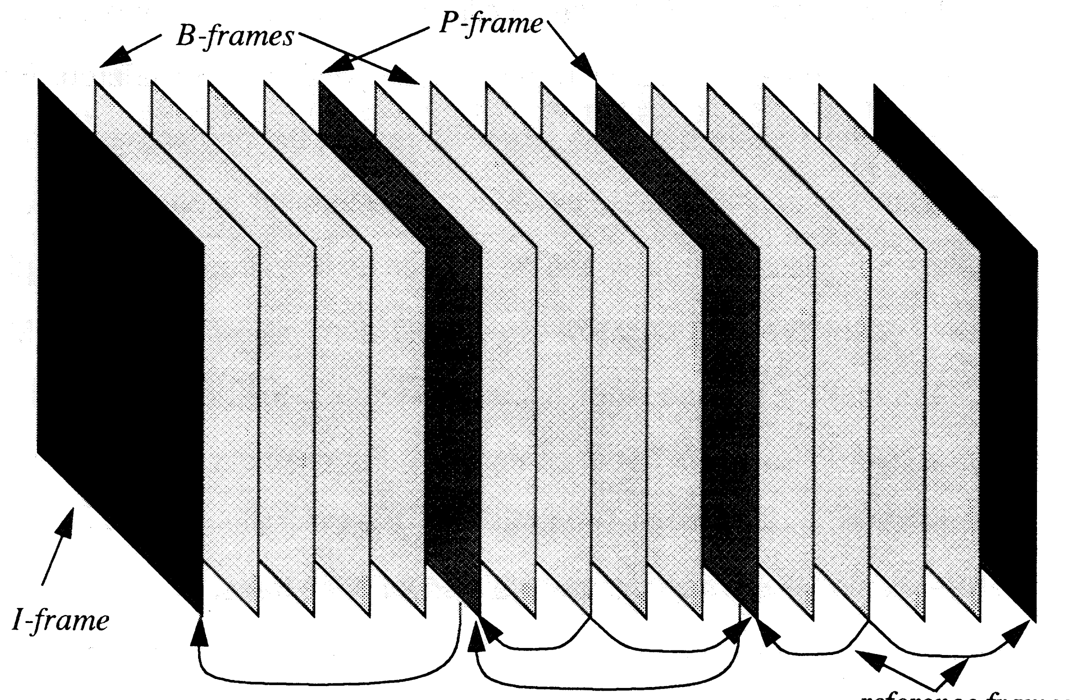


Figure 5-1: MPEG inter-frame dependencies

Now, consider the problem of transmitting an MPEG encoded video stream on an unreliable network using a transmission protocol that performs no error correction for lost packets. If all packets are equally likely to be lost in transmission, the number of packets required to send the frame determines the probability of a frame being received at the destination. Consequently, small frames will be more likely to get through, since they need fewer packets for transmission. Since reference frames are large, they are more likely to be dropped on an unreliable network.[1] But if an I-frame is lost, the dependent P- and B-frames are undecodable. Similarly, if a P-frame is lost, all subsequent B- and P-frames up to the next I-frame are undecodable.

A media-independent protocol transmits all frame types with equal priority, while a media-specific protocol can give higher priority to I- and P-frames, resulting in better playback at the destination. Prioritization can be achieved, for example, by allowing retransmissions of reference frames.

The advantages of media-specific protocols become even more apparent when the problem of reducing network congestion is considered. For example, suppose the system detects a large increase in the percentage of packet losses and attempts to compensate by reducing the number of packets sent. If the packets are not chosen carefully, the results can be disastrous. For example, a media-independent protocol might decide not to send an I-frame in an MPEG video stream, making the sequence undecodable up to the next I-frame. A better solution is to drop frames in dependency order. For example, an MPEG-specific protocol reported in [67] drops B-frames first, then P-frames, and finally the I-frame. More exotic MPEG-specific protocols could even transform P-frames to I-frames in the file server, possibly using compressed domain processing techniques based on the methods presented in chapter 3, if sufficient CPU resources are available. Clearly, such transformations cannot be done in a media-independent way.

[1] Typical frame sizes for a 352x240 MPEG compressed video are 12 KBytes for I-frames, 8 KBytes for P-frames, and 1 KBytes for B-frames. Assuming 1.5 KByte packets, an I-frame takes about 8 packets to transmit, a P-frame takes about 6 packets, and a B-frame takes about 1 packet.

Media-specific protocols can be built on a variety of underlying transmission or network protocols. For example, the MPEG media-specific protocol described above can be built on TCP/IP. The problem with using TCP/IP is that the implementation is difficult because TCP/IP provides the wrong features. For example, if the network becomes briefly congested, the appropriate action is to dequeue those frames that will arrive too late for playback. But most TCP/IP implementations will not allow data to be removed from a TCP/IP stream once the data is queued for delivery.

Work-arounds can be used to solve these problems, but a better solution is to use a protocol with the features needed to build media-specific protocols. A transmission protocol that provides these features, called *cyclic-UDP*, is described in section 5.2. Cyclic-UDP is designed to satisfy the needs of media-specific protocols and to work well on current local and wide area networks. This latter point is demonstrated by experimental results presented in section 5.4 that characterize the properties of cyclic-UDP in both local and wide area network (LAN and WAN) environments. These experiments show that high quality video playback is possible on the Internet today. For example, in one experiment, a 352x240 full-color, MPEG video sequence and associated audio was sent from U.C. Berkeley to Cornell University, across eighteen gateways and 2800 miles, at about 17 frames per second. Of course, such performance is limited to a small number of simultaneous connections and by the bandwidth of the links on the Internet.

The basic idea of cyclic-UDP is to give high priority packets a better chance of delivery by allowing more retransmissions of these packets if they are lost. A consequence of this strategy is that it requires buffering about four seconds of data at the receiver, a small amount of memory (typically under one MByte) on today's computers.

The rest of this chapter is organized as follows. Section 5.2 describes three protocols I tested for inclusion in CMT. Section 5.3 develops media-specific protocols for MPEG, motion JPEG, and audio data. Section 5.4 reports the results of experiments designed to test the effectiveness of these media-specific protocols in typical applications and environments and their effect on non real-time traffic. And the last section compares the best-effort protocols developed in this chapter with other CM protocols.

5.2 Protocol Suite

This section describes three protocols that I evaluated for use in CMT: 1) Simple-UDP, 2) UDP+resends, and 3) cyclic-UDP. These protocols are currently implemented on UDP, the user datagram protocol.

Simple-UDP

To discuss the characteristics of transmission protocols, it is useful to have a straw man against which to compare them. Our straw man is a protocol called *simple-UDP*. In simple-UDP, the CMS process sends timestamped frames of CM data to the CMX process a short time (typically about 0.5 seconds) before they are required for playback. By sending frames early and buffering them at the destination, the effect of network jitter on playback is virtually eliminated; in modern networks, any frame not delivered in 0.5 seconds is almost certainly lost. Each frame is timestamped with two values, the logical start and end time, specifying the earliest and latest possible logical playback time, respectively. When the frame is received, it is queued for playback by calculating the system times corresponding to the timestamp window and creating an at-event to call the playback function in this time window as described in chapter 4.

The biggest problems with simple-UDP are 1) it has no flow control and 2) it does not resend lost packets. When a network connection is overloaded, the

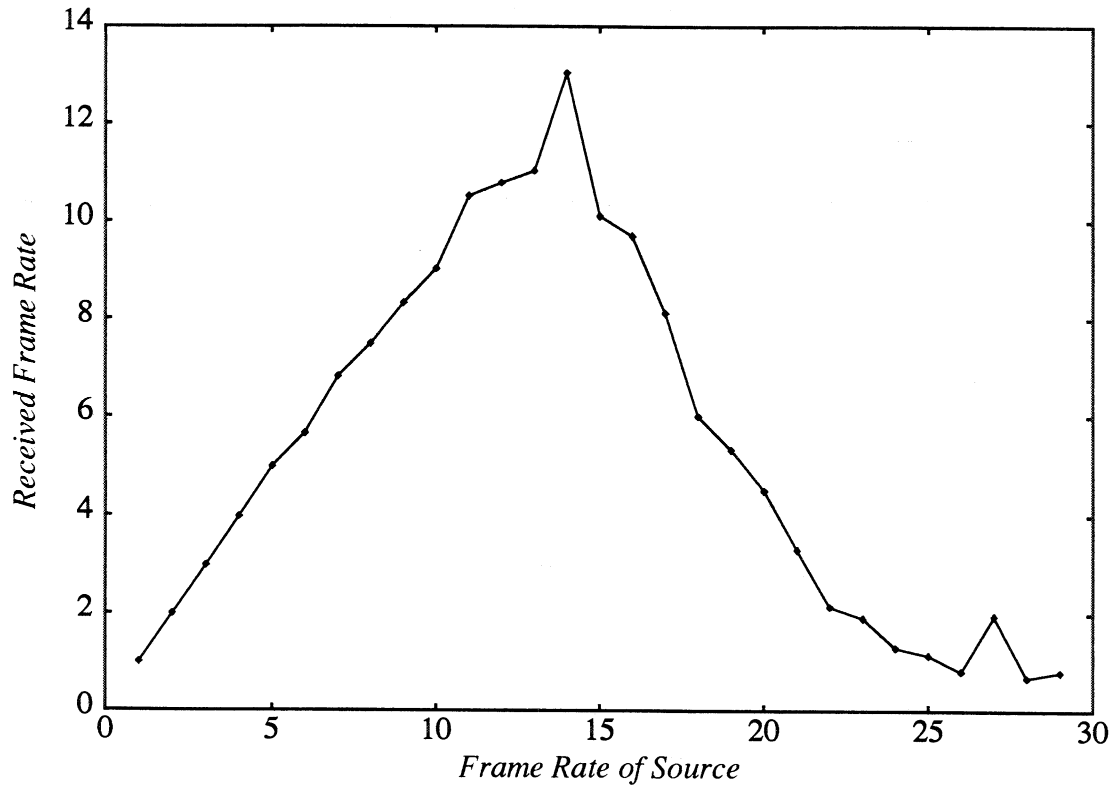


Figure 5-2: Frame rate at source vs. receiver on a long haul network

probability of packet loss increases dramatically due to buffer overflows on the routers. Because frames are fragmented into several packets, the probability of a frame being successfully reassembled at the destination is small. For example, figure 5-2 shows the probability of receiving a video frame when a fixed frame rate is used to deliver video on a typical long haul network. The average size of a video frame in this experiment was about 12 KBytes and each frame was sent as a single UDP datagram. The horizontal axis shows the frame rate of the sender in frames/sec, and the vertical axis shows the frame rate of the receiver in frames/sec. This particular connection (between Cornell University and UC Berkeley) overloaded at about 14 frames/sec, after which the probability of receiving a frame plummets. The well-known solution to this problem is to use some form of flow control on the channel: when losses get too high, the application should throttle back on the load it is placing on the network.

The second thing to notice in simple-UDP is that it sends fairly large messages (e.g., the size of one video frame). These large messages are usually fragmented into several packets for transmission, and if one packet is lost, the whole frame is discarded. A retransmission scheme that allows the lost packet to be resent will improve throughput, as shown by the next protocol.

UDP+Resends

UDP+resends was the first CM protocol used in CMT [68]. UDP+resends provides fragmentation, limited retransmission, and flow control. A *CM Connection* (CMC) is established between the CMS process and CMX process that consists of two channels: a data channel and a control channel. The data channel carries CM data and uses the UDP protocol. The control channel carries flow control and retransmission information and uses the TCP protocol.

In addition to timestamping frames and sending them early as in simple-UDP, UDP+resends also fragments the frame into one or more packets, each of which contains at most 8 KBytes of frame data. A *packet* is constructed from a fragment by prepending a *packet header* and storing the packet in a circular buffer known as the *packetQueue*. The size of the packetQueue is chosen so that the buffer containing the packet will be recycled no earlier than its playback time.

The packet header contains the fields listed in table 5-1. *StartTime* and *endTime* indicate the logical time window for frame playback, as in simple-UDP. *FrameNumber* specifies the sequence number of this frame. When a CMS is started, the *frameNumber* is set to zero and incremented for every frame queued for delivery. *FrameSize* stores the total size of the frame in bytes, and *packetSize* stores the size of the fragment contained in this packet. *PacketNumber* indicates the fragment number of this packet, *numPackets* indicates the total number of fragments for this frame, a nd *packetOffset* is the byte offset of the fragment from the

Field	Purpose
startTime	Logical Start Time of Frame Playback Window
endTime	Logical End Time of Frame Playback Window
frameNumber	A monotonically increasing frame identifier
frameSize	The frame size
packetNumber	The packet number within the frame
numPackets	The number of packets in this frame
packetOffset	The byte offset of the packet's data from the beginning of the frame
packetSize	The number of bytes in this packet

Table 5-1: Contents of UDP+resends packet header

beginning of the frame.

When the destination receives a packet, it copies the packet data into a circular buffer, detects lost or duplicate packets, and issues a Tcl-DP RPC request to the CMS process to retransmit missing packets. When a frame is reassembled, an at-event is created to play the frame as in simple-UDP.

Two modules, the *PacketSource* and the *PacketDest*, implement UDP+resends in CMT. When the PacketSource is given a frame for transmission (by the MediaSource, described in section 4.3), it fragments and queues the frame. The fragments are sent to the CMS in a series of bursts. In each burst, up to *burstSize* bytes are sent consecutively to the destination, starting with the oldest packet in the packetQueue that is marked as unsent. The first burst is scheduled as an idle-event. Subsequent bursts, if necessary, are scheduled as at-events that fire after a delay of *burstPeriod*. *BurstSize* and *burstPeriod* are 35 KBytes and 10 milliseconds, respectively, in the current system, which allows a maximum throughput of 26 Mbits/sec for each stream. These values represent an empirically determined compromise between sending the data in large bursts, which is easy on the application process since it reduces the number of system calls and context switches, and sending the data in small bursts, which minimizes the chance of transiently overloading the network by metering the injection of new data.

Within the CMX process, the *PacketReceived* callback within the PacketDest module is invoked when a packet arrives. PacketReceived copies the frame fragment into the appropriate location in a circular buffer of frames. Duplicate packets are dropped, and a list of missing packets is periodically sent to the CMS as part of a Tcl-DP RDO call that marks the packets in the CMS packetQueue as unsent. The time between these retransmission requests is called the *complainTime*, and the RDO calls are scheduled using at-events. When the PacketDest has completed the reassembly of the frame, it is passed to the MediaDest (described in section 4.3), which schedule playback.

UDP+resends uses a flow control mechanism that decreases the rate at which data is transmitted under two conditions: 1) when the system detects that frames are being dropped in the network (called *network frame drops*) and 2) when the CMX cannot decode and display all transmitted data (called *server frame drops*). Network frame drops typically occur during a congestion episode on the network. Server frame drops typically occur during software decoding because the CPU is not fast enough to play the stream at the full data rate or because the machine is executing a CPU-intensive process [67]. In either case, the CMX sends a *backoff* message to the CMS process telling it to reduce the rate of CM data transmission.

Like the missing packet resend request, the backoff message is sent as a Tcl-DP RDO over the control channel. The backoff message contains a *penalty factor* that is used for flow control as follows. Each stream has two parameters, FPS_{min} and FPS_{max}, that specify the minimum acceptable frame rate measured in frames per second (*fps*) and the maximum usable fps. For video, FPS_{min} is typically 15 and FPS_{max} is typically 30. The penalty factor is a value between zero and one that is used to calculate the stream's current FPS_{out}. FPS_{out} specifies the number of frames that the source will send to the destination each second. FPS_{out} is calculated from FPS_{min} and the current FPS_{out} using the following formula:

$$FPS_{out} = (1 - penalty) \bullet FPS_{out} + penalty \bullet FPS_{min} \qquad \text{EQ 5-1}$$

Penalty is calculated as follows. When a frame is lost due to network problems, penalty is increased by 0.1. A penalty of 0.1 is also assessed for each frame that is received but misses its play window. An additional penalty of 0.1 is assessed for each consecutive frame that misses its play window. For example, when three frames in a row miss their play window, the penalty is 0.5: a penalty of 0.1 for each frame that missed its window and an additional penalty of 0.1 for each consecutive frame that missed its window. If the total penalty exceeds 1.0, no more penalties are assessed. The total penalty, if non-zero, is transmitted every one-third of a second to the CMS, and the counting of penalties begins anew.

When the backoff message is received, FPS_{out} is adjusted as specified above. In addition, FPS_{out} is increased by one every *rampTime* seconds up to FPS_{max}. Ramptime is typically one second. This linear increase in FPS_{out} is scheduled as an at-event in the CMS process.

This flow control scheme decreases the rate at which data is transmitted from source to the destination aggressively when congestion is detected, and slowly increase it towards FPS_{max} when the congestions abates. This scheme is similar to the way window control in TCP/IP works: the window size decreases exponentially when losses are detected and increases linearly over time.

Figure 5-3 illustrates the behavior of the flow control scheme. The horizontal axis is time, and the vertical axis is fps, with FPS_{min} and FPS_{max} marked. The two lines on the graph show FPS_{out} and $FPS_{received}$, the fps received by the

CMX process. At the point marked A in the graph, several frames have been lost, as can be seen by the dip in $FPS_{received}$, and a backoff message with a penalty factor of 0.5 is sent to the CMS process. The backoff message causes the FPS_{out} to dip after which it slowly climbs back towards FPS_{max}. At point B, a backoff message with a smaller penalty factor is sent to the CMS.

Although UDP+resends overcomes the worst shortcomings of simple-UDP, its congestion control method is slow to respond and can lead to poor quality play-back when the network is highly congested. Congestion control is slow to react to backoff messages because the backoff messages change the rate at which data is read from the disk, not the rate at which data is fed into the network. This leads to a delay before responding to the backoff message. The delay is proportional to the difference between a frame's playback time and the time it is read from disk.

The second problem with UDP+resends is that the design makes it hard for the CMS to deliver multiple streams because frames are read off the disk independently by each MediaSource, causing the disk to thrash if each read incurs a large seek. For example, suppose an average disk seek takes 10 milliseconds

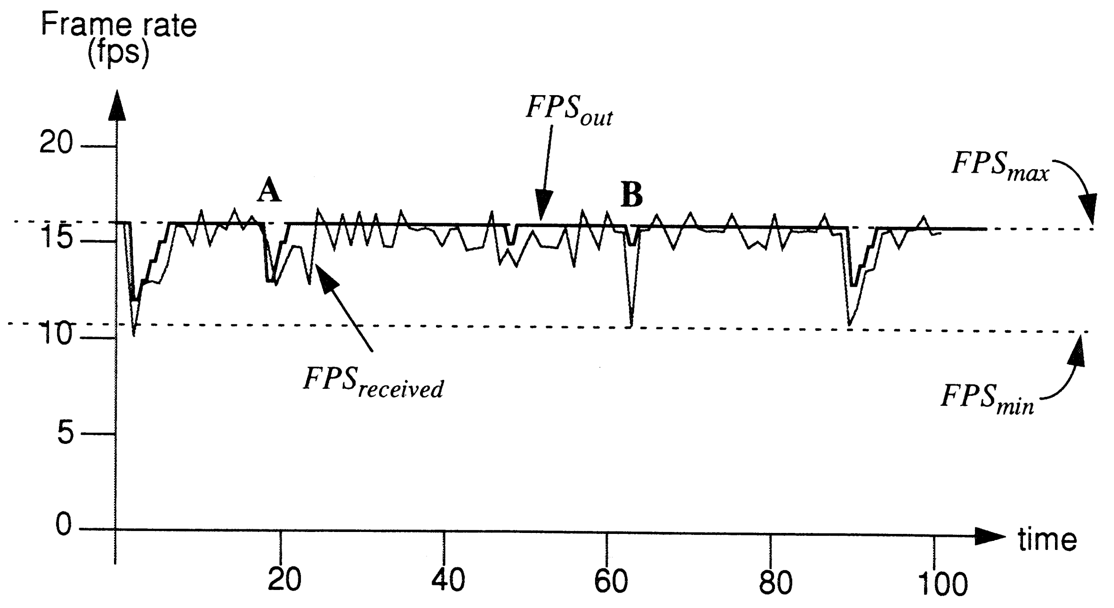


Figure 5-3: UDP+resends adaptive flow control

and the data transfer rate is 2 MByte/second. Then reading a 12 KByte frame (a typical value) can take up to 16 milliseconds, so the system can only read 60 frames per second. Hence, a source can support only two streams per disk which is much too inefficient. Reading larger blocks of data is the obvious solution.

The protocol discussed next, called *cyclic-UDP*, provides a solution to both problems.

Cyclic-UDP

Cyclic-UDP is a transmission protocol that delivers a prioritized set of *buffers* of CM data (containing, for example, a compressed video frame or a group of audio samples) from the CMS process to the CMX process as quickly as possible. Prioritization, in this context, means that buffers with higher priority are more likely to be delivered than buffers with low priority, given sufficient time, all buffers will be delivered. The advantage of prioritization is that it simplifies the implementation of media specific protocols.

Prioritization is very important in cyclic-UDP. In fact, one of the primary contributions of this work is showing how prioritized buffer delivery can be achieved on unreliable networks using adaptive flow control techniques. I will show later in this chapter that a high QOS can be maintained using this transport protocol along with media specific prioritization.

In cyclic-UDP, a MediaSource reads a group of frames from a local disk, determines the priority order of the frames using media-specific protocols discussed later in this chapter, and passes this prioritized list of frames to the PacketSource, which fragments and sends them to a PacketDest. The PacketDest reassembles the frames and passes them to a MediaDest, which queues the frame for playback.

Our concern is how a group of buffers is transmitted from a PacketSource to a

PacketDest. For the following discussion, assume the end-to-end bandwidth on a connection, called the *estimated bandwidth* (*estBW*) is known. The determination of estBW is discussed below.

In cyclic-UDP, a group of buffers are sent during a single *cycle*. A new cycle is begun when the MediaSource sends a *NewCycle* message to the PacketSource that includes a list of buffers (i.e., the frames from the MediaSource), highest priority first, as a parameter. The PacketSource constructs packets from the buffers by fragmenting each buffer, attaching a packet header to each fragment, and placing the resulting packets in a queue maintained by the PacketSource called the *packetQueue*. Cyclic-UDP is designed so that packets near the front of packetQueue have a higher probability for delivery than packets near the end of the packetQueue.

The contents of the packet header are listed in Table 5-2. The fields can be broken into four groups: fields identifying the packet and cycle, fields used for fragmentation and reassembly, fields used for measuring connection properties, and the EOC field.

In the identification group, *packetId* is an integer specifying the position in the packetQueue. *CycleId* identifies the cycle to which this packet belongs. *CycleId* is incremented by one with each new cycle except in special cases discussed below. *PacketID* and *cycleID* serve as a unique identifier for this packet. *BurstId* is an integer, set when the packet is transmitted, specifying the burst number. Lastly, *PPC* (packets per cycle) and *BPC* (buffers per cycle) contain the number of unique packets and buffers, respectively, that will (ideally) be delivered during this cycle.

In the fragmentation group, *bufferId* identifies the buffer to which this packet belongs (*bufferId* begins at zero at the beginning of each send cycle), *bufferSize* is the size of the buffer from which the packet was constructed, and *ppb* is the number of unique packets required to reassemble this buffer. *PacketOffset* is the byte offset of the fragment within its buffer; it is included to simplify reassembly.

In the connection measurement group, *sendTime* is the value on the sending machine's system clock when the packet is queued for delivery (used by the receiver to calculate network delay), and *epochNumber*, *epochBytes*, and *epochDuration* are used to calculate the connection bandwidth and loss rate.

Finally, *EOC* is set if this packet is an end-of-cycle (EOC) marker. The use of EOC packets is discussed later.

Field	Contents
packetId	Unique id for this packet
cycleId	Cycle number this packet is part of
burstId	Burst number when packet was sent
ppc	Packets in this cycle (pkts/cycle)
bpc	Buffers in this cycle (buffers/cycle)
ppb	Packets in this buffer (pkts/buffer)
bufferSize	Total size of buffer (in bytes)
bufferId	unique ID for this buffer
packetOffset	Offset of this packet's data in its buffer
sendTime	Time packet was sent
epochNumber	Epoch number when packet was sent
epochBytes	Number of bytes sent in the previous epoch
epochDuration	Length of previous epoch (msecs)
EOC	non zero means this packet marks the end

Table 5-2: Cyclic-UDP packet header

Packet transmission takes place in a series of bursts. During a burst, packetQueue is scanned in priority order, and packets marked as *unsent* (all packets are initially marked as *unsent*) are prepared for transmission by setting *burstID* in the packet header to the *current burst number*. The current burst number is a small integer that starts at zero at the beginning of each send cycle and increases by one for each burst. The packet is then sent to the destination as a UDP datagram and marked as sent.

The number of bytes queued for delivery in each burst is set so that the average bandwidth is estBW and no more than *maxBurstSize* bytes are sent in a single chunk to prevent overfilling the destination buffer. If there are no unsent packets in packetQueue before this many bytes are sent, an *End-of-cycle* (*EOC*) packet is sent. In either case, a new call to `SendBurst()` is scheduled using an at-event. The time of the at-event is calculated such that the next send burst is of size *burstSize* (the ideal number of bytes to send in a burst) plus the unused bandwidth from this burst. If every packet in packetQueue is marked as sent, the delay is set to 100 milliseconds. Figure 5-4 shows the pseudo-code for this process.

When a packet is received, the buffer fragment is copied into the correct position in memory. The implementation is careful to avoid data copies by using the bufferId and packetOfs fields in the packet header to determine the correct buffer and the position within the buffer. Lost packets are detected by examining the packetId field in the header. If the current packet is not the next packet in sequence, the missing packets are presumed lost[2].

When a lost packet is detected a *resend request* is sent to the CMS process

[2] I abandoned other, more complex schemes for loss detection after experiments showed that out-of-order packets are rare, even on long-haul connections.

```
SendBurst()
   burstNumber = burstNumber + 1;
   bytesSent = 0;
   currTime = ReadSysClock();
   p = 0;
   bytesToSend = min (maxBurstSize, (currTime - lastSendTime)*estBW);
   while (p<numPackets AND bytesSent < bytesToSend) {
      if (!packetQueue[p].sent) {
         packetQueue[p].sent = 1;
      packetQueue[p].burstId = burstNumber;
      SendPacket (packetQueue[p]);
      bytesSent += packetQueue[p].packetSize;
      }
      p = p + 1;
   }
   delay = (burstSize + bytesSent - bytesToSend)/estBW;
   if (p == numPackets) {
      SendEOC (cycleNumber, burstNumber, numPackets);
      delay = 0.1;
   }
   lastSendTime = currTime;
   after (delay, SendBurst);
```

Figure 5-4: Sending a burst of packets

requesting retransmission of all missing packets from the current cycle. The resend request contains the cycleId of the missing packets and a list of {*burstId, packetId*} pairs. *BurstId* is the burst identifier from the packet header received when the lost packet was first detected, and the *packetId* identifies the lost packet. Figure 5-5 shows the pseudo-code for detecting missing packets and constructing a resend request.

Each resend request is sent to the CMS as a UDP datagram. When a resend request is received by the CMS, the `ProcessResend` function shown in figure 5-6 is called. `ProcessResend` compares the *burstId* of the resend request with the *burstId* of the packet in the packet queue. If the latter *burstId* is greater than the *burstId* in the resend request, the request is stale because the packet has been resent since the resend request was issued, and the request can be ignored. Otherwise, the packet is marked as *unsent* and will be retransmitted dur-

```
DetectLoss(pkt):
   if (pkt.packetId > lastPkt+1) {
      for p = lastPkt+1 to pkt.packetId-1 {
         missingBurstId[p] = pkt.burstId;
         missing[p] = TRUE;
      }
      resendReq.cycle = pkt.cycleId;
      resendReq.resendList = {};
      for p = 0 to pkt.packetId-1 {
         if (missing[p]) {
            element.packetId = p;
            element.burstId = missingBurstId[p];
            append element to resendReq.resendList;
         }
      Send(resendReq);
      }
   }
   missing[pkt.packetId] = FALSE;
   lastPkt = pkt.packetId;
```

Figure 5-5: Detecting missing packets in the CMX process

ing the next call to SendBurst.

For example, suppose the *packetQueue* contains five packets. Figure 5-7 shows the transmission sequence. The tables below the figure show the relevant information in the *packetQueue* at the times indicated. Initially (time T_0), all packets are marked as unsent and the *burstId* of each packet is zero. In the first burst, packets one and two are sent; accordingly, the *burstId* of each packet is set to one, the packets are sent to the PacketDest in the CMX process, and the packets

```
ProcessResend(resendReq):
   if (resendReq.cycle == currCycle) {
      foreach element in resendReq.resendList {
         b = element.burstId;
         p= element.packetId;
         if (pktQueue[p].burstId <= b AND pktQueue[p].sent) {
            pktQueue[p].sent = FALSE;
         }
      }
   }
```

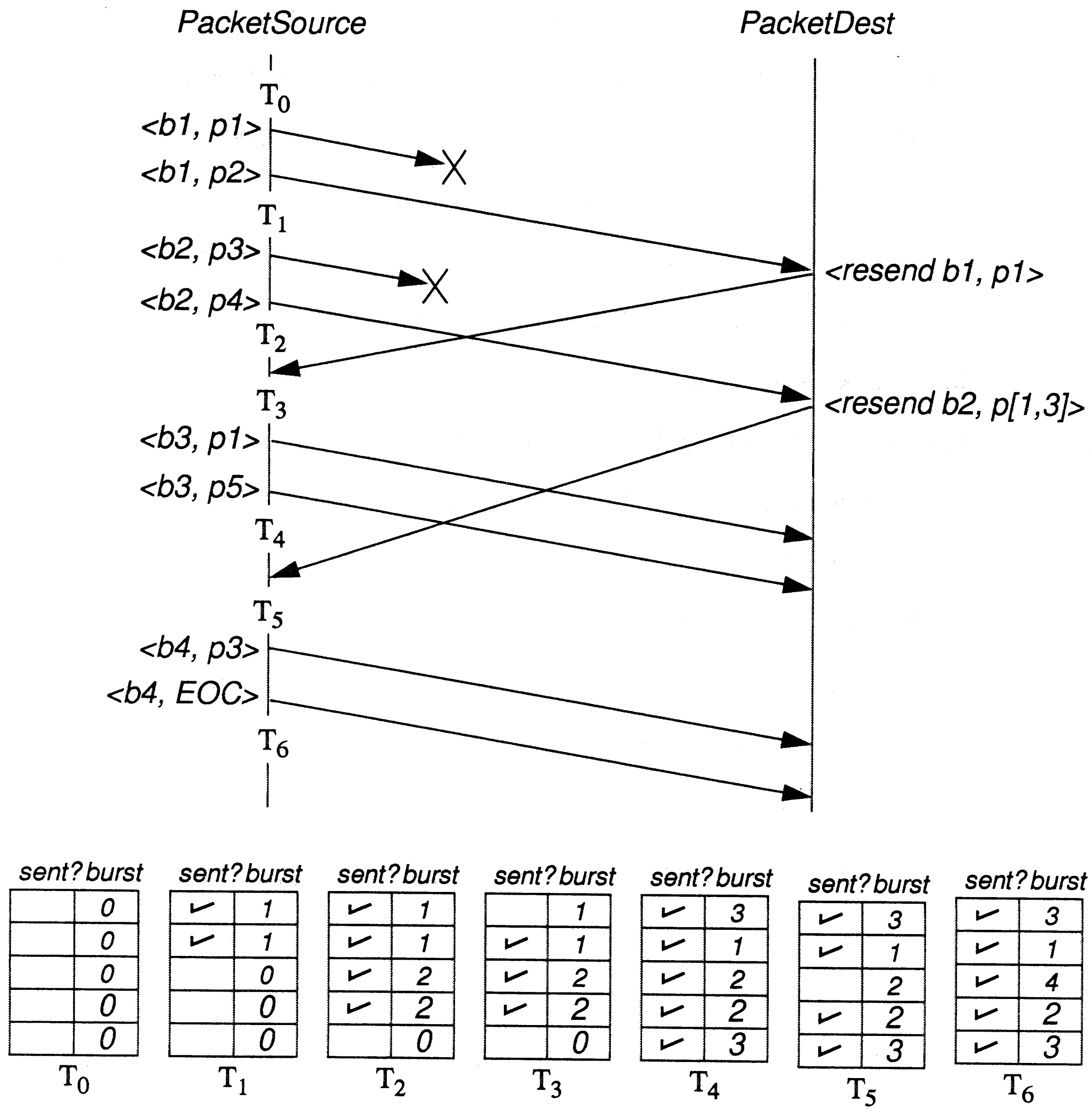
Figure 5-6: Source side resend processing

are marked as *sent* (time T_1). Now suppose packet one is lost and the CMX receives packet two. CMX marks packet two as received, and sends a resend request for packet one with the *burstId* equal to one (denoted *<resend b1, p1>*). Meanwhile, the PacketSource sends packets three and four in the second burst (time T_2), but packet three is lost. Shortly thereafter (time T_3), the PacketSource receives *<resend b1, p1>* and marks packet one as *unsent*. When the PacketDest receives packet four, it issues a resend request for both packets one and three: denoted *<resend b2, p[1,3]>*. During the next send burst (between T_3 and T_4), packet one and packet five are marked as *sent* and transmitted, with the *burstId* of each set to three. The PacketSource then receives the second resend request, but ignores the resend of packet one since the *burstId* of the request is two, but packet one was resent during burst three.Only the request for packet three is valid, so packet three is marked as *unsent* and is sent out in the next burst, along with an end-of-cycle (EOC) packet.

The EOC packet is a special packet used to detect lost packets near the end of the packetQueue. An EOC packet contains no frame data and has its EOC field set to 1. It is sent when all packets in the packetQueue have been marked as *sent*, and thereafter once every 100 milliseconds. On receiving an EOC, the receiver issues a resend request for all missing packets in the current cycle.

To see the use of the EOC packet, consider the above example, and suppose the last packet (packet five) is lost. When the CMX receives the EOC packet, it can detect that packet five was lost by examining the EOC's *ppc* field, which equals five in this example. Even if the EOC packet is lost, an EOC will be sent every send burst, and one will eventually make it through, triggering the resend request for packet five.

When a new send cycle begins, the cycle number is incremented. The CMX detects this change when a packet arrives whose *cycleId* is larger than what was



T0 sent?	T0 burst	T1 sent?	T1 burst	T2 sent?	T2 burst	T3 sent?	T3 burst	T4 sent?	T4 burst	T5 sent?	T5 burst	T6 sent?	T6 burst
	0	✓	1	✓	1		1	✓	3	✓	3	✓	3
	0	✓	1	✓	1	✓	1	✓	1	✓	1	✓	1
	0		0	✓	2	✓	2	✓	2		2	✓	4
	0		0	✓	2	✓	2	✓	2	✓	2	✓	2
	0		0		0		0	✓	3	✓	3	✓	3

Figure 5-7: Cyclic-UDP example

previously received. When the new cycle is detected, the packets that are part of incomplete buffers are moved into a *previous cycle* buffer, the resend list is cleared, and the packets that were in the previous cycle buffer are marked as free. Any packets arriving late that are part of the previous cycle are correctly reassembled as part of the previous cycle; should a buffer be completely reassembled, it is scheduled for playback by the MediaDest.

CycleIds can also be used to manage network buffers in the CMX. If the *cycleId* is increased by two, the CMX assumes it lost *all* packets of the intermediate cycle, and the buffer space associated with both the previous and current cycles is marked as free. The *cycleId* jumps when a discontinuous change occurs in the logical time system, such as when the user randomly seeks to a new position in the video. In this case, the PacketDest flushes all previously received buffers to avoid confusion.

Cyclic-UDP gives high priority packets near the front of the queue a better chance of getting through because, in the event of packet loss, they will get more retransmission requests and will be sent more times than packets later in the queue. If the cycle length is several times the round trip time through the network, a packet near the front of the queue will get many chances to get through the network. During a period of high packet loss, low priority packets (near the end of the packetQueue) may not be transmitted, since the high priority packets will get several retransmissions, but given sufficient time, all buffers will eventually be delivered to the destination.

Flow Control

Cyclic-UDP uses an estimate of the end-to-end available network bandwidth (*estBW*) for flow control. *EstBW* is adaptively computed by the protocol using measurements made by the CMX process. The values measured are listed in table 5-3.

The computation of *meanDelay* and *delayDeviation* are similar to what is used in TCP [75]. The only difference is that TCP estimates round trip delay whereas cyclic-UDP measures end-to-end delay. I use the following scheme to estimate end-to-end delay. When each packet is transmitted, the sendTime field of the packet is set using the value on the sender's system clock. When the packet is

Field	Contents
recvdBW	The bandwidth at the receiver
loss	Percent of packet loss
meanDelay	Mean packet delay
delayDeviation	Mean deviation of packet delay

Table 5-3: Measurements made by the CMX process

received, this value is subtracted from the value on the receiver's system clock. The minimum value of this difference in recent history is assumed to be the skew on the system clock. The skew is subtracted from the difference to get the measured delay, *M*. *MeanDelay* and *delayDeviation* are computed from *M* using the following formulas (taken directly from [75]):

$$err = M - meanDelay \qquad \text{EQ 5-2}$$

$$meanDelay' = meanDelay + \frac{err}{8} \qquad \text{EQ 5-3}$$

$$delayDev' = delayDev + \frac{|err| - delayDev}{4} \qquad \text{EQ 5-4}$$

Loss and *recvdBW* are computed over a period known as an *epoch* which is typically 100-200 milliseconds. During each epoch, the source tracks how many bytes it transmits and the destination tracks how many bytes it receives (*bytesRecvd*). When the epoch ends, the sender sets the *epochBytes* and *epochDuration* fields of each packet in the new epoch to the total number of bytes sent by the source and the duration of the epoch (in milliseconds), respectively.

When the receiver detects a new epoch (by examining the *epochNumber* field on a received packet), *loss* and *recvdBW* are computed using

$$loss = 1 - \frac{bytesRecvd}{epochBytes} \qquad \text{EQ 5-5}$$

```
Measure(pkt):
1    delay = ReadSysClock()-pkt.sendTime;
2    if (delay < skew) skew = delay;
3    delay -= skew;
4    err = delay - meanDelay;
5    meanDelay += (err >> 3);
6    delayDev += ((abs(err) - delayDev) >> 2);

7    if (pkt.epochNumber > currEpoch) {
8      if ((pkt.epochNumber == currEpoch+1) AND (pkt.epochBytes != 0))
9        bw = 1000*bytesRecvd/pkt.epochDuration;
10       loss = 100 - 100*bytesRecvd/pkt.epochBytes;
11       Feedback (bw, loss, meanDelay, delayDev, currEpoch);
       }
12     currEpoch = pkt.epochNumber;
13     bytesRecvd = 0;
     }

14   if (pkt.epochNumber == currEpoch) bytesRecvd += numBytes;
```

Figure 5-8: Destination Measurements of connection

$$recvdBw = \frac{bytesRecvd}{epochDuration} \qquad \text{EQ 5-6}$$

After computing *loss* and *recvdBw*, the destination transmits the current values of *meanDelay*, *delayDeviation*, *recvdBW*, and *loss* to the source in a *feedback unit* sent as a UDP datagram. The code in figure 5-8 summarizes the measurement process. A few subtle points in the code are the detection of skipped epochs, where *every* packet in the intervening epochs was lost (line 8) and the detection of late arriving packets (line 14).

The PacketSource uses the feedback units sent by the destination to set the *estBW* parameter used for flow control. The PacketSource uses two user parameters, the expected loss rate (X) and the target delay (D_t), along with the data in the feedback unit to calculate *estBW* and the long term estimate of the maximum bandwidth available on the connection, *ltBW*. *LtBW* is updated using a long-lived weighted average (with gain α, typically 1/64) of the bandwidth in the feedback unit:

$$err = recvdBW - ltBW \qquad \text{EQ 5-7}$$

$$ltBW' = ltBW + \alpha \bullet err \qquad \text{EQ 5-8}$$

Since *ltBW* is an estimate of the maximum long term bandwidth on the connection, this formula is only evaluated when either the loss is non-zero (which means the connection is being pushed to its limit) or the bandwidth contained in the feedback unit is greater than the current value of *ltBW* (which means the current value of *ltBW* is low).

Once the *ltBW* has been updated, *estBW* is computed using the following scheme. If the loss at the receiver is greater than the expected loss (*L>X*), *estBW* is set to $(1 - L) \bullet ltBW/(1 - X)$, which scales *estBW* linearly with loss above the expected loss. If the loss is low ($L < X$), but the delay at the receiver is greater than the target delay ($D > D_t$), *estBW* is set to $(1 + X) \bullet ltBW(D_t/D)$, which causes *estBW* to drop quickly as the delay increases. Otherwise, *estBW* is set to $(1 + X) \bullet ltBW$. The rationale in this case is that the bandwidth should be set to the maximum channel bandwidth plus some extra bandwidth proportional to the loss rate. Finally, if the result is outside the range [minBandwidth..maxBandwidth], *estBw* is set to either *minBandwidth* or *maxBandwidth* to guarantee that the bandwidth never rises above what is needed or sinks below a threshold. The code in figure 5-9 implements these strategies.

5.3 Media specific protocols

This section shows how media specific protocols use prioritization. Three media types are examined: motion JPEG video, MPEG video, and uncompressed audio. When used in combination with a transport mechanism that provides prioritized delivery (e.g., cyclic-UDP), these protocols reduce the effect of transient

```
ProcessFeedback (feedback):
   if ((feedback.loss != 0) OR (feedback.bandwidth > ltBW)) {
      err = feedback.bandwidth - ltBW;
      ltBW += err>>alpha;
   }

   estBW = (1+X)*pktSrc->ltBW;
   if (feedback.loss > X) {
      estBW = estBW*(1-feedback.loss);
   } else if (feedback.delay > Dt) {
      estBW = estBW*Dt/feedback.delay;
   }
   estBW = min (maxBandwidth, estBW);
   estBW = max (minBandwidth, estBW);
```

Figure 5-9: Adjusting the bandwidth based on feedback

load to a decrease in fidelity to the end user. The strategies described here are implemented in CMT in the MediaSource object.

Before discussing the packet delivery algorithms, it will be useful to define two concepts: inverse binary ordering (IBO) and playback jitter. The IBO of a group of N objects is obtained by reversing the bits in the binary representation of the object number and sorting the result. Object numbers start at zero. For example, the IBO of a group of four objects numbered {0, 1, 2, 3} is {0, 2, 1, 3}, and the IBO of a group of 11 objects numbered {0..10} is {0, 8, 4, 2, 10, 6, 1, 9, 5, 3, 7}. The advantage of IBO is that if the tail of a sequence in IBO is cut off, the lost objects are evenly distributed in the original sequence. For example, if the last 5 objects in the 11 element IBO above are lost, every other object in the original sequence is lost. The IBO will be used in the media-specific protocols that follow.

Playback jitter is defined as the standard deviation of the delay between the display time of sequential video frames. Playback jitter (measured in milliseconds) provides a quality metric, since users are sensitive to erratic frame drops. In other words, playing 15 fps may be better than playing 20 fps if the 15 frames are played at an even stride (low playback jitter) while the 20 frames are played with jerky intervals (high playback jitter). Playback jitter is calculated over a one second period because users notice erratic effects in about that time scale.

To gain an intuition for the meaning of playback jitter, consider a playback sequence where two frames are played and one is dropped, two more are played and another is dropped, and so on. Such a playback sequence can be represented diagrammatically as xx-xx-xx-xx-xx-xx-..., where each x represents a played frame and each - represents a missed frame. The diagram x-x-x-x-x, therefore, represents a sequence where every other frame is played and the jitter is zero. Table 5-4 lists the playback jitter for several frame sequences, assuming

Sequence	FPS	Delta (msec)	Jitter (msec)
X-X-X-X-X-X-X-X-X-X-X-X-	15	67	0
XX-XX-XX-XX-XX-XX-XX-XX-	20	50	17
XX-X-XX-X-XX-X-XX-X-XX-X-	23	44	17
X---XXXX-X-XXX--X--XXX-	17	59	33
XXX---XXX---XXX---XXX---	15	67	45
XXXXXX------XXXXXX------	15	67	75

Table 5-4: Jitter for various playback sequences

each sequence is to be played at 30 fps (33.3 milliseconds between frames). The mean time between frames (*delta*) and the corresponding frame rate are also listed. The first three sequences are fairly smooth, the next three are increasingly bursty. A reasonable rule to summarize the concept of playback jitter is this: if the playback jitter is less than half the interframe play time (i.e., delta), the sequence will look fairly smooth. If the playback jitter is more than delta/2, the sequence will appear bursty. A possible direction for future work is to conduct a human factors study to formally test the relationship of playback jitter to perceived quality.

I will now describe media-specific protocols for three media encodings: motion JPEG video, uncompressed audio, and MPEG video.

Motion JPEG

In motion JPEG video streams, each frame can be decoded independently of other frames. To make motion JPEG streams robust against packet loss, groups of frames in a read/send cycle are prioritized using IBO. For example, suppose that a read/send cycle is 0.5 seconds long and the frame rate is 30 fps. The fifteen frames are prioritized using IBO, which leads to the following prioritized frame list (PFL), highest priority first: {0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7}. If the frames are received in priority order, the playback jitter of the resulting sequence is small.

Uncompressed Audio

Uncompressed audio streams are composed of a set of samples taken at regular intervals. The rate of sampling is called the sampling rate. Typical sampling rates vary from 8 kHz for telephone quality audio to 44 kHz for CD quality audio. Because the individual samples are small (typically one to two bytes), each network packet contains between 20 and 500 milliseconds worth of samples.

To make audio streams robust against packet loss, the following media specific protocol is used. Each group of audio samples is subsampled into packets of approximately 2000 bytes, and each packet is independently delivered to the destination. For example, a 500 millisecond CD audio frame will contain 22000 samples. This frame is split into 11 packets of 4400 samples each. The first packet contains samples numbered $11i$ (i is an element of [0..1999]), the second packet contains samples numbered $11i + 1$, and so on up to the fourth packet. The effect of this subsampling is to make each packet contain a low fidelity version of the frame. The receiver reconstructs the frame by interpolating missing packets. Packets are prioritized in IBO to make the interpolation as accurate as possible.

MPEG

In MPEG video streams, some frames are encoded differentially (P-frames and B-frames) and some independently (I-frames). A dependency graph can be constructed for the frames that defines a partial ordering that is used to determine the priority of a frame in a group of frames: a reference frame should always have higher priority than its dependent frames. Although this condition must be satisfied by any good media-specific MPEG ordering algorithm, it does not uniquely define the order.

For example, consider the 16 frame MPEG sequence IBBBPBBBPBBBPBBB. This sequence, whose dependency graph is shown in figure 5-10, is denoted by $\{I_0, B_1, B_2, B_3, P_4, B_5, B_6, B_7, P_8, \ldots\}$ where the frame number has been appended to each frame. One possible prioritized frame list (*PFL*) for this group of frames is $\{I_0, P_4, P_8, P_{12}, I_{16}, B_2, B_6, B_{10}, B_{14}, B_{11}, B_1, B_5, B_{15}, B_9, B_3, B_{13}, B_7\}$. Table 5-5 gives all the possible playback patterns generated by this PFL, assuming frames are received in priority order. As the table shows, this PFL does well at minimizing playback jitter.

To generate the PFL, the basic operation needed is to decide which frame not already in the PFL to add next. The selected frame should clearly be *eligible* to be sent. That is, all its reference frames should already be elements of the partial PFL. Using the notation of table 5-5, the selected frame should break up runs of

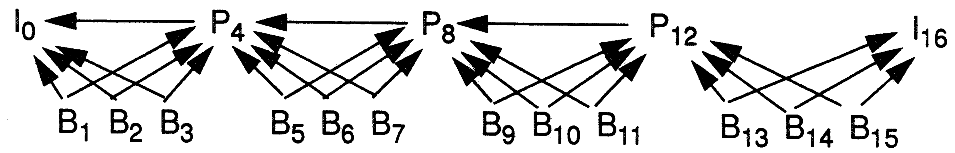


Figure 5-10: Inter-frame dependencies in sample MPEG sequence

# Frames Received	Pattern	# Frames Received	Pattern
1	X----------------	10	X-X-X-X-X-XXX-X-X
2	X---X------------	11	XXX-X-X-X-XXX-X-X
3	X---X---X--------	12	XXX-XXX-X-XXX-X-X
4	X---X---X---X----	13	XXX-XXX-X-XXX-XXX
5	X---X---X---X---X	14	XXX-XXX-XXXXX-XXX
6	X-X-X---X---X---X	15	XXXXXXX-XXXXX-XXX
7	X-X-X---X-X-X---X	16	XXXXXXX-XXXXXXXXX
8	X-X-X---X-X-X-X-X	17	XXXXXXXXXXXXXXXXX
9	X-X-X-X-X-X-X-X-X		

Table 5-5: Playback patterns of PFL for sample MPEG sequence

dashes (*white-chains*) as much as possible, since doing so minimizes the jitter of the playback subsequence. This requirement is formally expressed by minimizing the expression

$$W^2 - L^2 - R^2 \qquad \text{EQ 5-9}$$

where W is the length of the white-chain to which the frame belongs before the split, L is the length of the white-chain to the left of the frame after the split, and R is the length of the white-chain to the right of the frame after the split.[3] Once all the long white chains are eliminated (after step 9 in table 5-5), our goal is then to minimize the length of runs of X's (*black-chains*) generated by adding the new frame. For example, between steps 15 and 16 in the example, both frames are eligible, but the right frame is preferred since the new black-chain it forms is shorter than the black-chain formed when the left frame is added.

[3] The length of a white-chain is the number of dashes in the chain plus one.

Having sketched the ideas of how to choose the next frame, I now turn to the task of designing an efficient algorithm to compute the PFL. The input to the algorithm is an array of frames. For each frame, the information listed in table 5-6 is stored in an array of structures called the *ChainArray. URight, uLeft, wcRight*, and *wcLeft* store the edges of the white-chain and the black-chain to which the frame belongs, *dependents* stores a list of other frames in the input that use this frame as a reference frame, and *refCount* indicates the eligibility of the frame. *RefCount* initially contains the number of frames in the input sequence that are reference frames for this frame. Thus, all I-frames have a *refCount* of zero, most P-frames have a *refCount* of one, and most B-frames have a *refCount* of two. The only exceptions are when the P- or B-frames use a reference frame not in the input array. The code in figure 5-11 initializes the *dependents* and *refCount* fields of the *chainArray*. The other values are initialized as specified in table 5-6. After initialization, a heap [71] that contains the eligible frames is built so that the item at the top of the heap is the next frame to send. The heap is sorted using the following criteria, in order of importance:

```
ref = -1;
for (i=0; i<numberOfFramesInInput; i++)
   if ((ref != -1) && (frame[i].type != 'I')) {
      List_Append (frame[ref].dependents, i);
      frame[i].refCount++;
   }
   if (frame[i].type != B) ref = i;
}
ref = -1;
for (i=numberOfFramesInInput-1; i>= 0; i--) {
   if (frame[i].type != 'B') ref := i;
   else if (ref != -1) {
      List_Append (frame[ref].dependents, i);
      frame[i].refCount++;
   }
}
```

Figure 5-11: Initializing the `ChainArray`

Name	Description	Initial Value
refCount	If zero, frame is eligible; if negative, frame has been sent. If positive, frame is ineligible.	Number of frames in input that are reference frames for this frame
uRight	Index of nearest unsent frame to the right of the frame.	(frame index) + 1
uLeft	Index of nearest unsent frame to the left of the frame.	(frame index) - 1
wcRight	Index of right edge of the white-chain to which this frame belongs.	(number of input frames) - 1
wcLeft	Index of left edge of the white-chain to which this frame belongs.	0
dependents	List of frames that use this frame as a reference frame.	List of dependent frames. See figure 5-11

Table 5-6: The `ChainArray` data structure used for MPEG prioritization.

1. Minimizing the black-chain length (given by uRight-uLeft)
2. Maximizing the white-chain length (given by wcRight-wcLeft)
3. Minimizing the reduction in jitter (given by equation 5-9).
4. Minimize the frame size.

Criteria number 3 is evaluated using *wcRight*, *wcLeft*, and the index of the frame i to compute W, R, and L in equation 5-9:

$$W = wcRight - wcLeft \qquad R = wcRight - i \qquad L = i - wcLeft$$

The algorithm removes the item at the top of the heap (the *best* node), adds it to the PFL, and performs the steps in figure 5-12 to update the ChainArray. This code updates *uRight, uLeft, wcRight*, and *wcLeft* and decrements the *refCount* of each of the frames that use the selected frame as a reference frame. Since some of these frames may be eligible, these steps may affect the heap ordering. The `FixHeap()` function checks the *refCount* of the specified frame to see if it is eligible and, if so, moves it up or down in the heap as required to restore heap order.

```
node[best].refCount--;
for (i=node[best].wcLeft; i<best; i++) {
   frame[i].wcRight = best-i;
   FixHeap(i);
}
for (i=best+1; i <= node[best].wcRight; i++) {
   node[i].wcLeft = best+1;
   FixHeap(i);
}
l = node[best].uLeft;
r = node[best].uRight;
if (r < numFrames) {
   node[r].uLeft = l;
   FixHeap(r);
}
if (l >= 0) {
   node[l].uRight = r;
   FixHeap(l);
}
for each d in node[best].dependent
   if (--node[d].refCount == 0)
      HeapInsert (d);
```

Figure 5-12: Updating the `ChainArray`

5.4 Experiments

This section describes a set of experiments performed using an implementation of cyclic-UDP. The purpose of these experiments is to evaluate the performance and behavior of cyclic-UDP in various scenarios. Important questions are:

1. Does cyclic-UDP deliver frames in priority order?
2. How well does it estimate and share the available bandwidth?
3. How does it scale to different networks (e.g., LAN and WAN)?
4. How does it impact other protocols, such as TCP?
5. What is the quality of the output for various media?

To answer these questions, three canonical movies were used in these tests: a low bandwidth 352x240 MPEG video and 8 kHz associated audio with a total bandwidth of about 1.2 Mbits/sec ("Ferris Wheel"), a 320x240, full color, medium quality motion JPEG movie ("Andre and Wally B."), and a 640x480, full color high quality motion JPEG movie and associated CD quality stereo ("Tony De Peltrie"). Table 5-7 lists properties of the video portion of these movies.

Movie	Bitrate (Mbits)	I-Frame Size (KBytes)			P-Frame Size (KBytes)			B-Frame Size (KBytes)		
		min	avg	max	min	avg	max	min	avg	max
Ferris	1.25	5.3	7.4	16.6	4.7	5.0	7.7	2.0	4.4	5.0
Andre	2.83	7.9	11.5	13.1	-	-	-	-	-	-
Tony	5.86	13.1	21.4	25.5	-	-	-	-	-	-

Table 5-7: Properties of the experimental video streams

Transmission was tested in three environments: a 10 Mbit/sec local area network (LAN), a metropolitan area network (MAN) with three subnets connected by two gateways, and a wide area network (WAN) with 18 gateways (an Internet connection between UC Berkeley and Cornell University). In each environment, one to four copies of each movie were sent simultaneously and the fidelity of the reconstructed streams was measured. The fidelity of audio is the reconstructed sampling frequency; the fidelity of video is the playback jitter and playback rate. The throughput of a simultaneous FTP transfer was also measured to determine the impact on non real-time traffic.

Local Area Network Experiment

Many parameters affect the performance of cyclic-UDP, including the duration of the read/send cycle and the epochs, the packet size, the target delay, the expected drop rate, and the minimum and maximum bandwidth for each stream. I used the values of the parameters listed in table 5-8 after considerable experimentation in the LAN environment. The audio stream is given priority over the

Parameter	Audio	Video
Cycle Length	4 seconds	2 seconds
Epoch Length	200 msec	200 msec
Packet Size	8000	8000
Expected Drop Rate	20%	2%
Target Delay	100 msec	50 msec
Maximum Bandwidth	48/264 KBytes/sec[a]	30 fps @ maxVidSize[b]
Minimum Bandwidth	16/88 KBytes/sec	10 fps @ maxVidSize
Burst Size	8 KBytes	8 KBytes
Maximum Burst Size	24 KBytes	24 KBytes

Table 5-8: Cyclic-UDP Parameters used in LAN tests

a. These values are six times the bandwidth required by the audio stream.

b. I.e., the bandwidth required to deliver the stream's largest video frame at 30 fps

video stream by setting its minimum and maximum bandwidth higher than necessary and by setting the expected drop rate of the audio higher than the expected drop rate of the video, which allows the audio stream to experience a higher loss rate before decreasing its output bandwidth. Note also that the length of the read/send cycle is 2 seconds, a value that may seem quite long. I set it to this length because experiments revealed that an ethernet can experience delays of up to 400 milliseconds under heavy load, and long cycles lead to smoother playback.

In the first experiment, a particular movie was selected for transmission and one to four copies of the movie were sent for sixty seconds between two to eight machines connected to a single 10 Mbit/sec Ethernet while a TCP/IP bulk transfer and ordinary departmental traffic proceeded in the background. The machines used in the test were: one Sparc 1, four Sparc 1+, one Sparc 10, one HP 730, and one HP 750. Separate machines were used for each source and destination pair so that the interaction of the protocols could be measured, rather than how they handled contention for machine resources at the source or the destination. Figure 5-13 shows the experimental configuration. The receiving processes were instrumented to log the frame number and cycle number of each frame when it was received.

Both the audio and video streams in the MPEG sequence "Ferris" were perfectly reconstructed, independent of the number of streams. This result in not surprising, since the aggregate bitrate required to deliver four simultaneous "Ferris" streams is about half of the network capacity.

The upper graph in figure 5-14 shows the probability of a video frame being received versus its priority[4] for the "Andre" video sequence. The different curves represent one, two, three, or four concurrent transports of "Andre." This graph shows that the probability of frame reception is a decreasing function of priority, as expected. With one or two video streams, almost all video frames in "Andre" came through every cycle. Only a few of the lowest priority frames were lost. Since each stream corresponds to about 29% of the Ethernet capacity and part of the capacity was used by departmental traffic, some frame loss when three or more concurrent streams are sent is inevitable. The graph shows that frames are correctly prioritized when loss does occur.

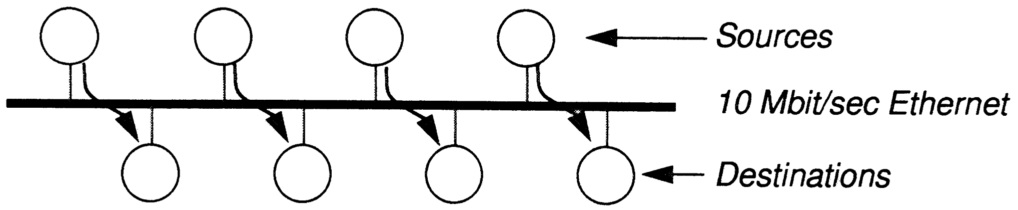
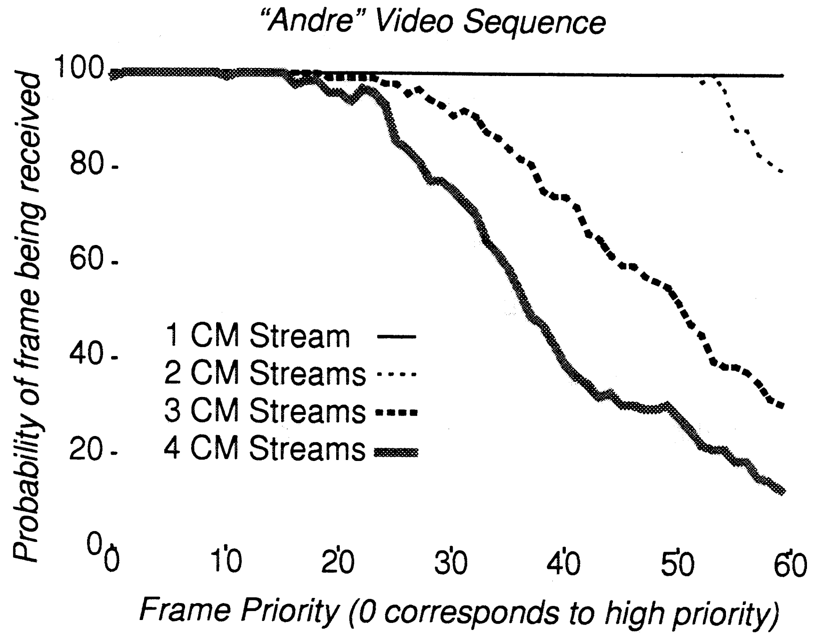


Figure 5-13: Configuration for LAN experiment

[4] Priorities are in the range 0-59, since the video cycle length is two seconds and the stream rate is 30 fps. Lower priority numbers correspond to higher priority frames.



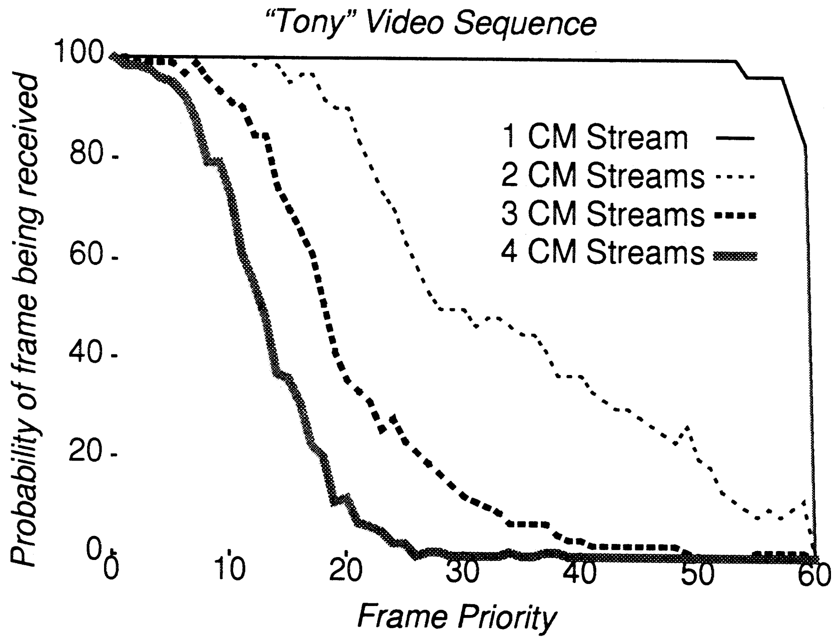


Figure 5-14: Probability of video frame reception on a LAN

The lower graph in figure 5-14 shows the corresponding graph for the "Tony" video sequence. Prioritization in this sequence also behaved well, demonstrating that the cyclic-UDP protocol behaves well even under fairly extreme conditions.

Table 5-9 shows the fidelity of the reconstructed audio streams in this experiment. One row is listed for each test, and the minimum and average reconstructed frequency (in kHz) are reported. As the table shows, audio was reconstructed perfectly in all but the most severe cases, and even then the average quality was very high.

Number of Movies	Ferris		Andre		Tony	
	Min	Avg	Min	Avg	Min	Avg
1	8.0	8.0	8.0	8.0	44.0	44.0
2	8.0	8.0	8.0	8.0	44.0	44.0
3	8.0	8.0	8.0	8.0	33.0	43.8
4	8.0	8.0	8.0	8.0	33.0	43.0

Table 5-9: Fidelity of reconstructed audio streams in a LAN

Movie	Ferris				Andre				Tony			
	FPS		Jitter		FPS		Jitter		FPS		Jitter	
	Min	Avg	Max	Avg	Min	Avg	Max	Avg	Min	Avg	Max	Avg
1	30.0	30.0	0.0	0.0	30.0	30.0	0.0	0.0	26.0	29.8	12.3	1.0
2	30.0	30.0	0.0	0.0	25.0	29.6	13.6	1.7	2.0	17.7	354	22.6
3	30.0	30.0	0.0	0.0	9.3	24.3	45.4	10.4	2.0	10.6	401	41.3
4	30.0	30.0	0.0	0.0	8.0	20.2	44.1	15.3	2.0	7.3	518	65.4

Table 5-10: Fidelity of reconstructed video streams in a LAN

Table 5-10 shows the fidelity of the reconstructed video streams. Fidelity is measured as the frame rate (in fps) and playback jitter (in milliseconds). Since the frame rate can vary significantly over the course of the video, frame rate is evaluated over one second intervals, and the minimum and average values are reported. As discussed in section 5.3, playback jitter is considered good when its value is less than half the inter-frame arrival time. The only time when the worst case jitter was not "good" was when more than one concurrent copy of the "Tony" sequence was sent. The relevant entries in the table are highlighted.

A separate, but interesting question is: what happens to non real-time traffic while these CM streams are being transmitted? Are they completely drowned out, or do they still get some appreciable bandwidth? Table 5-11 shows the throughput of a TCP connection that was run in the background while the experiment pro-

Movie	Ferris		Andre		Tony	
	Through put	% of unloaded	Through put	% of unloaded	Through put	% of unloaded
1	218	89%	158	65%	127	52%
2	210	86%	133	55%	51.3	21%
3	172	70%	58.6	24%	41.6	17%
4	105	43%	31.6	13%	11.3	5%

Table 5-11: Throughput of TCP connection (in KBytes/sec) during LAN playback

ceeded. With no CM Streams running, the connection had an average throughput of 323 KBytes/sec. In addition to listing the average throughput of the connection, the table also lists the percentage of the unloaded throughput. As you can see, the TCP connection was not adversely affected until several streams were playing.

Metropolitan Area Networks

The second suite of experiments tested cyclic-UDP in a Metropolitan Area Network (MAN) environment. The MAN used in these experiments connects UC Berkeley to the International Computer Science Institute (ICSI) in downtown Berkeley via two gateways[5]. The parameter values listed in table 5-12 were used in the MAN experiments. The parameters are the same as used in the LAN tests except for the highlighted items. As in the LAN tests, the audio stream is given pri-

Parameter	Audio	Video
Cycle Length	4 seconds	2 seconds
Epoch Length	200 msec	200 msec
Packet Size	1100	1100
Expected Drop Rate	20%	10%
Target Delay	100 msec	100 msec
Maximum Bandwidth	48/264 KBytes/sec	30 fps @ maxVidSize
Minimum Bandwidth	5/26 KBytes/sec	3 fps @ maxVidSize
Burst Size	4 KBytes	4 KBytes
Maximum Burst Size	12 KBytes	12 KBytes

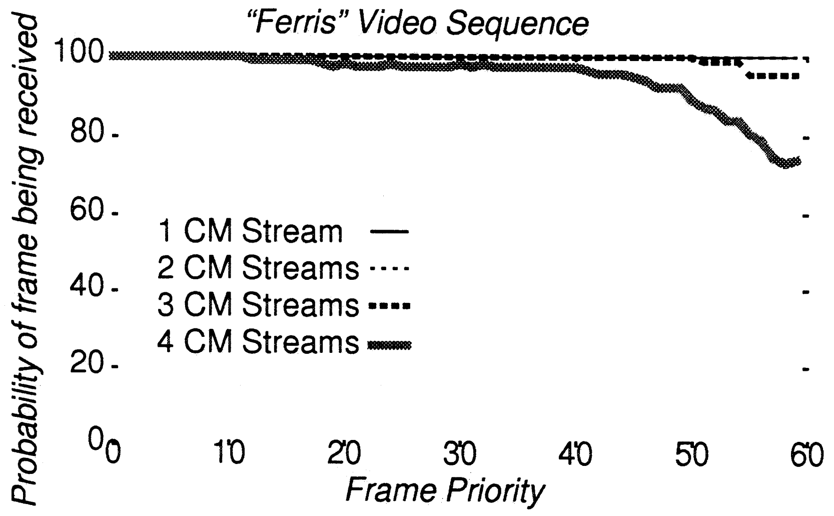
Table 5-12: Cyclic-UDP parameter in the MAN environment

ority over the video stream using the bandwidth and expected drop rate parameters.

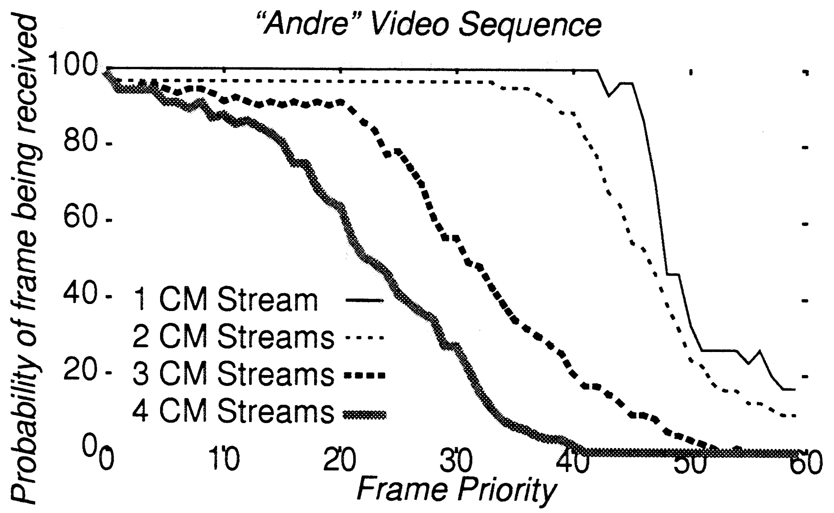
Figure 5-15 shows the probability of a video frame being received versus its priority for the various video sequences in the MAN environment. Again, cyclic-UDP prioritizes frames as expected. The MPEG sequence "Ferris" played perfectly for one and two concurrent streams, and nearly perfectly in all cases. "Andre" showed significant degradation when more than about two simultaneous streams were transmitted, and the quality of the "Tony" sequence dropped off sharply when more than one copy was sent. In all cases, however, the curves are generally decreasing, indicating that cyclic-UDP is prioritizing frames correctly.

Tables 5-13 and 5-14show the fidelity of the reconstructed audio and video streams. As in the LAN, audio fidelity was maintained even when video fidelity was poor. These experiments show that the MAN environment tested is capable

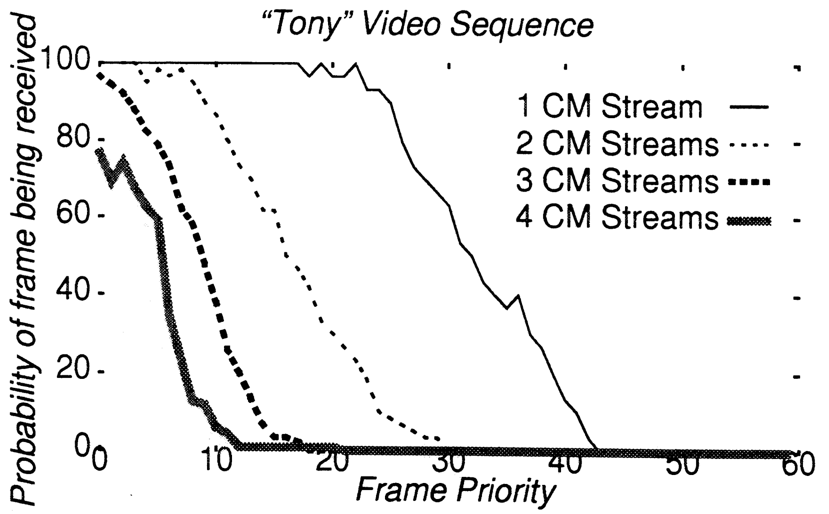
[5] The number of gateways along a route was determined using the `traceroute` utility, which uses the IP "time to live" field and attempts to elicit an ICMP TIME_EXCEEDED response from each gateway along the path.



(a)



(b)



(c)

Figure 5-15: MAN environment: Probability of video frame reception

Number of Movies	Ferris		Andre		Tony	
	Min	Avg	Min	Avg	Min	Avg
1	8.0	8.0	8.0	8.0	44.0	44.0
2	8.0	8.0	8.0	8.0	43.0	43.9
3	8.0	8.0	8.0	8.0	37.0	43.0
4	8.0	8.0	8.0	8.0	15.5	35.5

Table 5-13: Fidelity of reconstructed audio streams in a MAN

Movie	Ferris				Andre				Tony			
	FPS		Jitter		FPS		Jitter		FPS		Jitter	
	Min	Avg	Max	Avg	Min	Avg	Max	Avg	Min	Avg	Max	Avg
1	30.0	30.0	0.0	0.0	20.7	25.5	17.0	11.5	11.0	16.8	41.4	18.6
2	30.0	30.0	0.0	0.0	17.0	24.0	17.1	13.4	2.0	9.0	589	48.5
3	25.0	29.9	13.6	0.5	3.0	16.5	279	23.0	2.0	5.3	660	92.5
4	6.0	28.6	112	4.2	2.0	12.1	518	37.0	2.0	4.0	660	137

Table 5-14: Fidelity of reconstructed video streams in a MAN

of supporting about 5 Mbits/sec of CM traffic and that cyclic-UDP uses this bandwidth efficiently. Both "Andre" and "Tony" showed degradation when the total bandwidth exceeded this capacity, but only when the attempted bandwidth was four times the capacity was the average case jitter not "good."

Finally, table 5-15 shows the throughput of a TCP connection that was run in the background while the MAN experiment proceeded. With no CM Streams running, the connection had an average throughput of 73 KBytes/sec. As in the LAN case, the TCP connection was not adversely affected until several streams were playing. Note that cyclic-UDP obtains about ten times the throughput of TCP in this environment.

Movie	Ferris		Andre		Tony	
	Through put	% of unloaded	Through put	% of unloaded	Through put	% of unloaded
1	54	74%	36	49%	36	49%
2	41	56%	29	40%	20	27%
3	36	49%	36	48%	14	19%
4	20	28%	11	15%	4.5	6%

Table 5-15: Throughput of TCP connection (in KBytes/sec) during MAN playback

Wide Area Networks

The last set of experiments were designed to test cyclic-UDP in a Wide Area Network (WAN) environment. The WAN used was the Internet between UC Berkeley and Cornell University in upstate New York via eighteen gateways. The parameter values listed in table 5-16 were used in the WAN experiments. The

Parameter	Audio	Video
Cycle Length	4 seconds	2 seconds
Epoch Length	200 msec	200 msec
Packet Size	1100	1100
Expected Drop Rate	40%	20%
Target Delay	500 msec	100 msec
Maximum Bandwidth	120 KBytes/sec	30 fps @ maxVidSize
Minimum Bandwidth	12 KBytes/sec	1 KBytes/sec
Burst Size	4 KBytes	4 KBytes
Maximum Burst Size	12 KBytes	10 KBytes

Table 5-16: Cyclic-UDP parameter in the WAN environment

parameters are the same as used the previous tests except for the highlighted items. As in the previous tests, the audio stream is given priority over the video stream using the bandwidth and expected drop rate parameters.

Figure 5-16 shows the probability of a video frame being received versus its priority for the various video sequences in the MAN environment. In this experiment, we see some degradation in the MPEG sequence "Ferris". Also note that no attempt was made to send more than two concurrent copies of the "Tony" sequence across the Internet since the bandwidth required far exceeds the capacity of the channel and I did not want to affect other users too badly.

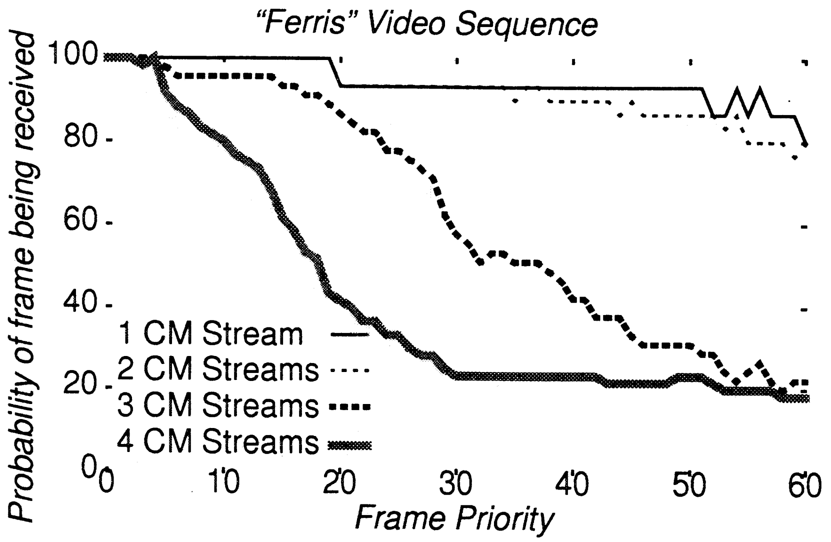
Tables 5-17 and 5-18 show the fidelity of the reconstructed audio and video

Number of Movies	Ferris		Andre		Tony	
	Min	Avg	Min	Avg	Min	Avg
1	8.0	8.0	8.0	8.0	21.0	27.6
2	8.0	8.0	8.0	8.0	11.5	26.3
3	8.0	8.0	8.0	8.0	-	-
4	7.0	8.0	6.0	7.9	-	-

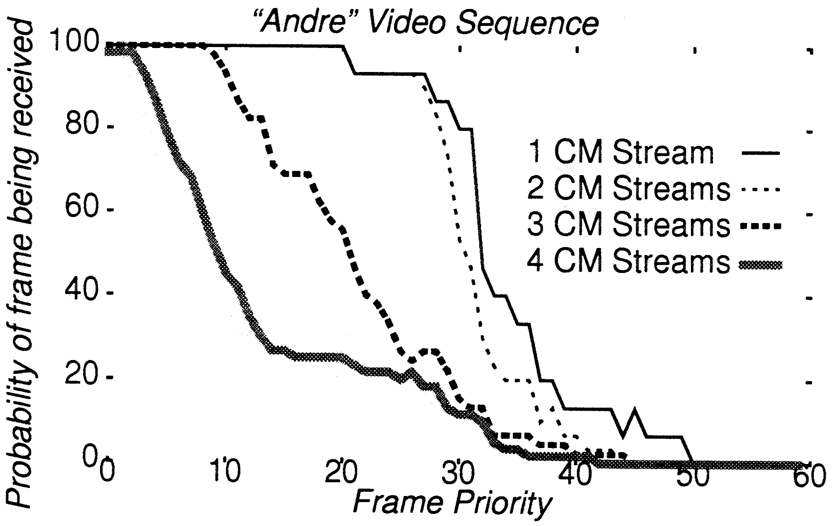
Table 5-17: Fidelity of reconstructed audio streams in a WAN

Movie	Ferris				Andre				Tony			
	FPS		Jitter		FPS		Jitter		FPS		Jitter	
	Min	Avg	Max	Avg	Min	Avg	Max	Avg	Min	Avg	Max	Avg
1	4.0	17.4	194	28.6	5.0	9.0	118	32.4	2.0	3.2	660	153
2	4.0	16.2	194	30.6	5.0	8.4	122	35.2	2.0	3.0	660	162
3	2.0	10.7	565	57.3	2.0	6.0	448	77.0	-	-	-	-
4	2.0	7.7	613	116	2.0	4.4	660	121	-	-	-	-

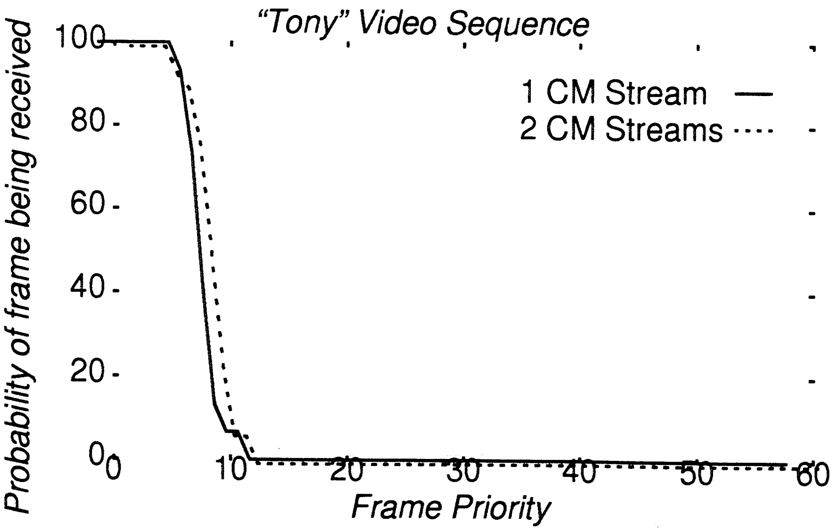
Table 5-18: Fidelity of reconstructed video streams in a WAN



(a)



(b)



(c)

Figure 5-16: Probability of video frame reception on a WAN

Movie	Ferris		Andre		Tony	
	Through put	% of unloaded	Through put	% of unloaded	Through put	% of unloaded
1	6500	51%	6060	48%	8430	67%
2	4300	34%	3000	24%	3680	30%
3	222	1.7%	1930	15%	-	-
4	1270	10%	3320	26%	-	-

Table 5-19: Throughput of TCP connection (in bytes/sec) during WAN playback

streams, and table 5-19 shows the throughput of a background TCP connection. All streams showed degradation in this environment in the worst case, but average case performance was acceptable except when four concurrent streams attempted to share this limited channel. The baseline connection throughput was about 13 KBytes/second, which corresponds to a frame rate of about 1 frame/second for the "Andre" sequence. Thus, cyclic-UDP achieves eight to eighteen times the throughput of TCP in this environment.

5.5 Previous Work

Previous work in network protocols for CM data delivery has been focused on solving three problems for applications: error and loss recovery, flow control, and delay and jitter management. Protocols and systems differ in how they address these problems. This section reviews the solutions to these problems that have been reported in the literature.

Error Recovery

In the literature, four basic techniques have been reported for packet loss and error recovery: 1) ignoring the error, 2) avoid the error, 3) retransmitting the corrupt/lost packet, and 4) correcting the corrupt/lost packet.

Many systems use strategy (1). Rather than recovering from errors, these systems initiate flow control when error rates exceed a threshold value in an attempt to reduce future errors. Jeffay and Stone [38] use this strategy in a system based on UDP, and the Heidelberg Transport System [33] uses this strategy, but on top of ST-II [77].

Nv [25], a video conferencing tool popular on the Internet, uses strategy (1) in combination with a custom compression technique that is robust to packet loss. In this technique, the source captures an image and breaks it into small blocks. Each block is compared against the corresponding block in the previous image and, if significantly different, is compressed and sent to the receiver(s). When the destination receives a block, it updates the portion of the screen that contains the block. Nv also periodically resends background blocks, even if they have not changed recently, just in case the original was lost in transmission.

The Tenet research group [22] has defined a protocol suite that provides statistical guarantees on network performance to implement strategy (2). Clients reserve resources in advance at nodes along the route between source and destination using the Real-Time Channel Administration Protocol, RCAP [5], specifying constraints on delay, jitter, and bandwidth. If the connection can be accommodated, it is accepted and a channel is established.

Cyclic-UDP uses strategy (3) and with prioritization to retransmit the optimal packets. Surprisingly few other systems use this paradigm, mostly due to fears that the latency will be too large for use in conferencing applications. TCP/IP [75] uses this paradigm; [31] reports efforts to extend TCP/IP to support multimedia applications by prioritizing packets in the routers, but this work is still embryonic. TCP/IP is the transport mechanism used in the Xphone system [20], a video conferencing prototype being developed at Columbia.

The idea of strategy (4) is to send redundant information that allows lost data to be reconstructed provided a subset of the original information is sent, a technique called forward error correction (FEC) [8,72]. Yavatkar and Manoj study two FEC strategies (XOR and replication) in a mutlicasting simulation study [85].

The Priority Encoded Transmission (PET) project [2], implements FEC in a particularly novel way. Each frame in a group of frames is assigned a numerical priority in the range (0..1], with smaller values yielding higher priorities. The group is then encoded into packets that are sent to the destination. The encoding has the property that, if a fraction x of the packets are received, then any frame with a numerical priority lower than x can be reconstructed.

Finally, the DEMON system supports strategies (1), (3), and (4), preprocessing the multimedia document to determine which strategy is most applicable to each data element in the document.

Flow Control

Virtually all best-effort systems provide some form of flow control in response to network congestion. Four strategies have been reported in the literature: 1) no flow control, 2) flow control based on measurements to estimate bandwidth, 3) flow control based on feedback messages from the destination, and 4) window-based flow control.

Systems that use no flow control typically use FEC or assume that flow control is provided by the network layer. For example, the Starlight Network Server [77] assumes that sufficient network bandwidth is available and the PET project [2] treats the network as a fundamentally lossy transmission medium where flow control is not necessary. The Tenet protocol [22] suite falls into this class, allowing applications to reserve bandwidth before they initiate transmission.

Cyclic-UDP falls into category (2), using measurements to estimate the available channel bandwidth. Jeffay and Stone allow multiple audio frames, but only a single video frame, to be queued at the sender. This strategy reduces the flow of video data in response to network congestion [38], dropping extra frames when access to the network is delayed.

UDP+resends uses strategy (3) to reduce the rate at which frames are generated at the source. HeiTP [19] uses media specific protocols to compensate for network congestion in response to feedback messages from the destination. And in the simulation study by Yavatkar and Manoj [85], several strategies for flow control based on destination feedback messages are investigated.

TCP/IP is the only system I know of that uses window-based flow control. The `Xphone` system [20], which sends audio and motion-JPEG data using TCP/IP, also uses window-based flow control by extension. But `Xphone` provides an additional layer of flow control using strategy (2) and a media-specific protocol, measuring the actual throughput every ten video frames and adjusting the quantization tables used in the JPEG compression accordingly.

Jitter management

The simplest solution to delay and jitter management is to buffer sufficient data at the destination to smooth out variations in network delay. The CMT system uses this approach. Although buffering is suitable for most playback applications, this solution is not viable for many interesting classes of applications. For example, conferencing applications need low end-to-end latency and can only buffer a maximum of about 800 milliseconds worth of data [64, 41, 10]. This implies that the network delay can not exceed 800 milliseconds. Furthermore, in some playback applications the receiver may have extremely limited buffering capabilities due to cost constraints. In both these situations, delay and jitter must be bounded.

The Tenet protocol suite allows applications to specify delay and jitter bounds at connection setup. In Xphone [20] and vat, the Internet audio conferencing tool, network jitter is absorbed at the application by dynamically adjusting the amount of data buffered at the destination and adjusting the delay between capture and display of data: if many samples are arriving after the playback point, more data is buffered and the delay is increased. If most samples are arriving early, buffering and delay are decreased. Both Xphone and vat reduce the buffering during periods of silence, so the change is almost invisible to the listener.

5.6 Conclusions and Future Work

In this chapter, I described an approach to the data delivery problem in multimedia systems. I showed that media-specific protocols, in combination with prioritized delivery mechanisms, can significantly improve the quality of playback systems. Cyclic-UDP is currently in use in CMT, and CMT has been delivered to several research groups outside Berkeley.

Some interesting extensions to this work are possible. For example, cyclic-UDP could be extended to conferencing applications. The biggest problem is that the notion of long read/send cycles is at odds with the low latency requirements of conferencing application. One approach to solve this conflict is to modify cyclic-UDP to dynamically add newly acquired frames to an existing cycle and to dynamically remove expired frames from a cycle. With these changes, the send buffer is like a TCP window on the stream of frames, except the window moves forward with time, not with ACKs as in TCP. This strategy introduces a new problem: how should the frames in the window be prioritized? One possibility is to make the window size fixed and send frames in bursts as in cyclic-UDP, prioritizing frames using IBO on the frame number.

For example, suppose the window size is four frames. Then the first frame is placed in slot one of the window, the second frame in slot three, the third frame in

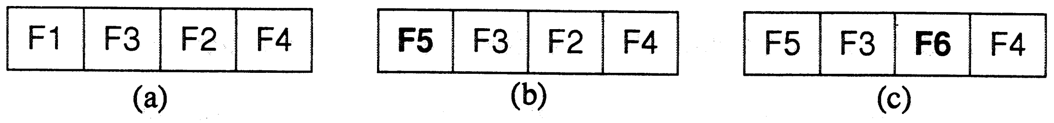


Figure 5-17: Extending Cyclic-UDP to Conferencing

slot two, and the fourth frame in slot four, as shown in figure 5-17 (a). Sending frames proceeds as in cyclic-UDP: at each burst, a fixed amount of unsent packets are sent, highest priority first. Retransmission requests mark packets as unsent. When a new frame is ready for sending, it is placed in the queue at the position determined by its frame number modulo the window size (four in this example). For example, figures 5-17 (b) and 5-17 (c) show the state of the window after frames five and six have been queued for delivery.

The interactive behavior of cyclic-UDP can also be improved. For example, if two second read/send cycles are used, the delay is two seconds from when the user presses *play* in a playback application and when video begins displaying on the screen. Although such delays may not be bothersome for viewing long video sequences, they are annoying if the user's behavior is interactive, as in browsing. Varying the cycle length in response to the user's interaction is a possible solution to this problem. The cycle length could be initially small and lengthened as the user's behavior becomes quiescent.

Chapter 6
Conclusions

This section enumerates the specific contributions of this thesis in the domain of multimedia systems and indicates directions for extending this work.

6.1 Compressed domain processing

Contributions

Chapters 2 and 3 described a novel, high performance technique for processing JPEG compressed images without decompressing them. The technique was based on the observation that the JPEG compression algorithm can be decomposed into a linear transformation (e.g., DCT + zig-zag scan + constant division) followed by a lossy, non-linear step (e.g., integer rounding) followed by an entropy coding step (e.g., Huffman coding). By writing the calculation of a digital video effect (DVE) as a linear transformation and composing it with the linear transformation of the JPEG algorithm, I showed how a large class of DVEs can be mapped into operators in the JPEG domain. By exploiting the approximation introduced by the lossy step in JPEG, I developed a technique called condensation that reduces the complexity of the JPEG domain calculation by several orders of magnitude. Two algorithms for condensation, thresholding and probabilistic, were also described and evaluated.

Applications

The DVE technology in chapters 2 and 3 has numerous applications, including:

1. **Transcoding**. These techniques can be extended to support real-time software video transcoding. For example, it might be possible to convert a motion-JPEG data stream to an MPEG

bitstream or an H.261 bitstream on demand and in software, a technology useful for database systems, video file servers, and video conferencing systems that support heterogeneous clients.

2. **Deinterlacing**. NTSC video is transmitted as a sequence of fields (designated odd and even) that contain every other line on the screen. Some capture systems [69] grab and compress video as a sequence of fields, whereas most applications, systems, and hardware store, process, and transport video as a sequence of frames (the composite of two fields). Consequently, the captured video fields must be converted to frames, a slow, off-line process. The DVE technology can be used to create deinterlacing software that quickly accomplishes this task.

3. **Editing**. Many current-generation editing systems use special purpose hardware to produce fades, wipes, dissolve, fly-ins, composition, and a variety of other effects in real-time. The DVE technology can be used to produce a software-only system with similar functionality at considerable savings.

4. **Source channel coding**. The data compression community has been exploring custom compression schemes that allow the bandwidth of the bitstream to be dynamically adjusted to provide network flow control. In these hybrid network/compression schemes, called source-channel coding, the frame rate, resolution, image quality, color depth, and inter-frame data dependencies are adjusted to modify the quality/bandwidth trade-off of the compression. Using the DVE technology, source-channel coding methods can be applied to standard encoding schemes.

Future Work

Although many applications listed above require research and development to be viable, extending this technology to new encodings is central to promoting widespread use. The basic ideas and techniques of chapters 2 and 3, namely mapping image processing operators into the compressed domain and condensing the result, can be applied to other transform based encodings, such are MPEG, H.261, sub-band, and wavelet encodings.

An intriguing path to explore is the application of these principles to other media types, such as audio. For example, MPEG audio compression uses sub-band coding along with a perceptual model to remove imperceptible data, so a condensation algorithm for MPEG compressed audio must integrate this perceptual model, a challenging problem.

6.2 CMT

Contribution and Applications

An orthogonal contribution of this thesis is a toolkit, called CMT, for constructing distributed CM applications like the CM Player. CMT is based on a modified version of the Tcl/Tk graphical user interface (GUI) toolkit and a distributed programming toolkit called Tcl-DP. The GUI toolkit modifications are generic; although I applied them to Tcl/Tk, the same ideas are applicable to almost any event-driven GUI.

Tcl-DP has received widespread attention in the Tcl community, and today hundreds of sites worldwide are using Tcl-DP to prototype client/server systems. It has been ported to most Unix systems, VMS, Windows 3.1, Chicago, UNICOS (the Cray operating system), and VxWorks. CMT has served as the basis for a Video-on-Demand System [21] and a video conferencing system [66]. Many researchers have expressed interest in using CMT for their own research, and efforts are underway to distribute a new version of the toolkit.

Future Work

CMT has great promise, but poses many challenges. The distributed aspect of the system makes it fragile; if a single process crashes, the system typically hangs. Standard techniques such as reliable backup servers may provide a solution to this problem, but clients will stall while the handoff to backup servers takes place. In future video file systems that support hundreds, or even thousands of users, the storm of reconfiguration requests will further delay the handoff.

One proposed solution to this problem is an extended video file server for CMT that stripes data at the frame level. For example, odd numbered frames might be stored on one server, and even numbered frames on another. If one server goes down, the frame rate is halved while backup servers are found. In other words, the effect of a server crashing is to reduce the fidelity of the stream.

6.3 Cyclic-UDP

Contribution

The final contribution of this thesis was a novel best effort network protocol for CM data delivery. This protocol differs from previous protocols in two ways. First, cyclic-UDP is designed to support media specific protocols, which I experimentally showed improve the quality of the reconstructed signal when the transport medium is unreliable. Media specific protocols and associated algorithms for MPEG, motion-JPEG, and uncompressed audio data were also presented.

The second way cyclic-UDP differs from previous work is that it works with a variety of encoding schemes and has been tested in LAN, MAN, and WAN environments. Other best effort protocols reported in the literature are either untested proposals [31, 9], have been tested only in one environment [68, 20, 38, 19, 76], have been simulated, but not implemented [85, 31, 9], or use custom compression schemes [15].

Future Work

Cyclic-UDP can be extended in two ways. First, cyclic-UDP can be extended to support the lower latency communication required by video conferencing applications. A method for implementing such an extension was proposed at the end of chapter 5, and will not be repeated here. Second, cyclic-UDP can be adapted to a high-speed cell network, such as ATM. I expect the protocol to behave well since it is highly asynchronous, but the question remains open until the protocol is actually implemented.

References

[1] A. Albanese, *personal communication*, 1994

[2] A. Albanese, J. Blomer, J. Edmonds, M. Luby, M. Sudan, *Priority Encoding Transmission*, Symposium on Foundations of Computer Science, October, 1994.

[3] D. P. Anderson, George Homsy, *A Continuous Media I/O Server and Its Synchronization Mechanism*, IEEE Computer, 1991 Oct, Vol. 24 Num. 10, pp. 51-57.

[4] F. Arman, A. Hsu, M. Y. Chiu, Image processing on compressed data for large video databases. Proceedings of ACM Multimedia 93, (Anaheim, CA, August 1-6, 1993). Association for Computing Machinery Press, New York, 1993 pp. 267-72.

[5] A. Banerjea, B. Mah, *The Real-Time Channel Administration Protocol*, Network and Operating System Support for Digital Audio and Video: Second International Workshop, (Heidelberg, Germany, November, 1992), Springer-Verlag, Berlin; New York. pp. 160-170

[6] J. Bates, *Presentation Support for Distributed Multimedia Applications*, Ph. D Thesis, Computer Laboratory, University of Cambridge, 1993.

[7] J. Bates, J. Bacon, *A development platform for multimedia applications in a distributed, ATM network environment*, Proceedings of IEEE International Conference on Multimedia Computing and Systems, Boston, MA, USA, May 15-19, 1994 IEEE Computer Society Press, Los Alamitos, CA, pp. 154-163.

[8] E. Biersack, *Performance Evaluation Of Forward Error Correction in an ATM Environment*, IEEE Journal on Selected Areas in Communications,

May 1993, Vol.11, Num. 4, pp. 631-640.

[9] J. Bolot, T. Turletti, *Feedback Control of Video Codecs for a Packet Switched Network*, Network and Operating System Support for Digital Audio and Video: Fourth International Workshop, Lancaster, UK, November, 1993. Springer-Verlag, Berlin; New York pp. 13-15

[10] P. Brady, *Effects of Transmission Delay on Conversational Behavior on Echo-Free Telephone Circuits*, The Bell System Technical Journal, Vol. 50, Num.1, January, 1971, pp. 115-134.

[11] M. C. Buchanan, P. T. Zellweger, *Automatic temporal layout mechanisms*, Proceedings of ACM Multimedia 93, (Anaheim, CA, August 1-6, 1993). Association for Computing Machinery Press, New York, 1993 pp. 341-350

[12] M. C. Buchanan, P. T. Zellweger, *Scheduling multimedia documents using temporal constraints*, Network and Operating System Support for Digital Audio and Video: Third International Workshop, (La Jolla, California, USA November 12-13, 1992). Springer-Verlag, Berlin; New York pp. 237-249.

[13] D. Bulterman, *Synchronization of Multi-Sourced Multi-media Data for Heterogeneous Target Systems*, Network and Operating System Support for Digital Audio and Video: Third International Workshop, (La Jolla, California, USA November 12-13, 1992). Springer-Verlag, Berlin; New York pp. 110-120

[14] Gordon Chaffee, *personal communication*, 1994

[15] S. Chakrabafti, R. Wang, *Adaptive Control for Packet Video*, Proceedings of the 1994 International Conference on Multimedia Computing and Systems, Boston, Mass, May 1994, pp. 56-62

[16] S. F. Chang, W. L. Chen, D. G. Messerschmitt, *Video Compositing in the DCT Domain,* IEEE Workshop on Visual Signal Processing and Communi-

cations, Rayleigh, North Carolina, Sept. 1992

[17] S. F. Chang and D. G. Messerschmitt, *A New Approach to Decoding and Compositing Motion Compensated DCT-Based Images*, IEEE International Conference on Acoustics, Speech, and Signal Processing, Minneapolis, Minnesota, pp. 421-424, April, 1993.

[18] B. Chitprasert, K. R. Rao, *Discrete Cosine Transform Filtering*, Signal Processing, Vol. 19, Num.3, pp. 233-245, March 1990

[19] L. Delgrossi, C. Halstrick, et. al., *Media Scaling for Audio Vidual Communication with the Heidelberg Transport System*, Proceedings of ACM Multimedia 93, (Anaheim, CA, August 1-6, 1993). Association for Computing Machinery Press, New York, 1993 pp. 99-104

[20] A. Eleftheridas, S. Pejhan, D. Anastassiou, *Algorithms and Performance Evaluation of the XPhone Multimedia Communication System*, Proceedings of ACM Multimedia 93, (Anaheim, CA, August 1-6, 1993). Association for Computing Machinery Press, New York, 1993 pp. 311-320

[21] C. Federighi, L. A. Rowe, *A Distributed Hierarchical Storage Manager for a Video-on-Demand System*, IS&T/SPIE Symposium on Electronic Imaging: Science & Technology, San Jose, California, February, 1994.

[22] D. Ferrari, A. Banerjea, H. Zhang, *Network Support for Multimedia - A discussion of the Tenet Approach*, Tech Report TR-92-072, Internation Computer Science Institute, Berkeley, CA, October, 1992; to appear in Computer Networks and ISDN Systems, special issue on Multimedia Networking, 1994.

[23] J. D. Foley, et. al., *Computer Graphics: Principles and Practice*, second edition, Reading, Mass. Addison-Wesley, 1990.

[24] J. D. Foley, A. Van Dam, *Fundamentals of Interactive Computer Graphics,*

Second Edition, Addison-Wesley Publishing Company.

[25] R. Frederick, *Experiences with Real-time Software Video Compression*, Proceedings of the Packet Video Workshop (Portland, Oregon, USA, September 26-27, 1994)

[26] D. R. Fuhrmann, et. al, *Experimental Evaluation of Psychophysical Distortion Metrics for JPEG-Encoded Image*, Proceedings of the SPIE - The International Society for Optical Engineering,1993, Vol.1913, pp. 179-190

[27] T. A. Funkhouser, C. H. Sequin, *Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments*, Proceeding of SIGGRAPH 93, Anaheim, CA, USA, August 1-6, 1993, Association for Computing Machinery, New York, NY, 1993. pp. 247-254.

[28] S. Gibbs, *Application Construction and Component Design in an Object-Oriented Multimedia Framework*, Network and Operating System Support for Digital Audio and Video: Third International Workshop, (La Jolla, California, USA November 12-13, 1992). Springer-Verlag, Berlin; New York pp. 351-355

[29] T. Grogan, *Image Quality Evaluation With A Contour-based Perceptual Model*, Proceedings of the SPIE - The International Society for Optical Engineering,1992, Vol.1666, pp. 188-197

[30] L. Hardman, D. Bulterman, G. Van Rossum, *The Amsterdam Hypermedia Model: Extending Hypertext to Support Real Multimedia*, Hypermedia, 1993, Vol.5, Num.1, pp. 47-69.

[31] B. Heinrichs, R. Karabek, *TCP/IP Supporting Real-Time Applications: The Predictive Delay Control Algorithm*, Network and Operating System Support for Digital Audio and Video: Second International Workshop, (Heidelberg, U.K., November, 1992), Springer-Verlag, Berlin; New York. pp. 45-47

[32] R. Herrtwich, *The HeiProjects: An Updated Survey*, Technical Report, IBM European Networking Center, Heidelberg, Germany October, 1992

[33] R. Herrtwich, L. Delgrossi, *Beyond ST-II: Fulfilling the Requirements of Multimedia Communication*, Network and Operating System Support for Digital Audio and Video: Third International Workshop, La Jolla, California, USA November 12-13, 1992, Springer-Verlag, Berlin; New York pp. 23-29

[34] A. Hopper, *Pandora - an experimental system for multimedia applications*, Olivetti Research Laboratory, Trumpington St., Cambridge, UK, CB21QA, June, 1990.

[35] B. K. P. Horn, *Robot Vision*, MIT Press, Cambridge, Mass, 1986

[36] E. Israel, *The X-Window system server: X version 11, release 5*. Digital Press, Bedford, MA, 1992.

[37] A. K. Jain, *Fundamentals of Digital Image Processing*, Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1989

[38] K. Jeffay, D. L. Stone, T. Talley, and F. D. Smith, *Adaptive, best-effort delivery of digital audio and video across packet switched networks*, Network and Operating System Support for Digital Audio and Video: Third International Workshop, (La Jolla, California, USA, November 12-13, 1992), Springer-Verlag, Berlin; New York, pp. 1-12,

[39] B. W. Kernighan, D. M. Ritchie, *The C Programming Language*, Englewood Cliffs, N.J., Prentice-Hall,1978.

[40] S. A. Klein, et. al. *Relevance of Human VIsion to JPEG-DCT Compression*, Proceedings of the SPIE - The International Society for Optical Engineering,1992, Vol.1666, pp. 200-215

[41] E. Klemmer, *Subject Evaluation of Transmission Delay in Telephone Conversations*, The Bell System Technical Journal, Vol. 46, July-August, 1967,

pp. 1141-1147.

[42] S. G. Kochan et. al, *UNIX Networking*, Hayden Books, Carmel, IN, 1989

[43] T. Lane, *Independent JPEG Group Software Codec*, Internet Software Distribution, URL ftp://ftp.uu.net/graphics/jpeg/jpegsrc.v4.tar.Z

[44] D. Le Gall, *MPEG: a video compression standard for multimedia applications,* Communications of the ACM, April 1991, Vol. 34, Num.4, pp. 46-58.

[45] J. S. Lim, *Two-dimensional signal and image processing*, Prentice Hall, Englewood Cliffs, N.J., 1990

[46] T. D. C. Little, A. Ghafoor, *Synchronization and Storage Models for Multimedia Objects*, IEEE Journal on Selected Areas in Communications, Vol. 8, Num.3, pp. 413-427, April 1990

[47] T. D.C. Little, D. Venkatesh, "*Probabilistic Assignment of Movie to Storage Devices in a Video-On-Demand System*," Network and Operating System Support for Digital Audio and Video: Fourth International Workshop, Lancaster, UK, November, 1993. Springer-Verlag, Berlin; New York pp. 213-224

[48] W. Mackay, G. Davenport, *Virtual Video Editing in Interactive Multimedia Applications*, Communications of the ACM, July 1989, Vol. 32 Num. 7, pp. 802-810

[49] D. Mills, *Measured performance of the network time protocol in the internet system*, Network Working Group, RFC 1128 (October 1988).

[50] T. J. Mowbray, T. Brando, *Interoperability and CORBA-based open systems*, Object Magazine, Sept.-Oct. 1993, Vol. 3, Num.3, pp. 50-54.

[51] *Multimedia System Services, Version 1.0*, Hewlett-Packard, IBM Corp., SunSoft, Inc. Response to Interactive Multimedia Association Request for

Technology, IMA Compatibility Project, 1993

[52] C. Nicolaou, *An Architecture for Real-Time Multimedia Communication Systems*, IEEE Journal on Selected Areas in Communications, Vol. 8, Num.3, pp. 391-400, April 1990

[53] N. B. Nill, *A Visual Model Weighted Cosine Transform for Image Compression and Quality Assessment*, IEEE Transaction on Communication, Vol. 33, Num.6, June 1985

[54] J. Ousterhout, *Tcl: An Embeddable Command Language*, Proc. of the 1990 USENIX Winter Conference, January 1990, pp. 133-146

[55] J. Ousterhout, *An X11 Toolkit Based on the Tcl Language*, Proc. of the 1991 USENIX Winter Conference, January 1991.

[56] *Parallax Video Software Developer's Guide*, Parallax Graphics, Inc. Santa Clara, CA. 1993

[57] W. B. Pennebaker, *JPEG still image data compression standard*, Van Nostrand Reinhold, New York, 1992.

[58] H. A. Peterson, *An Improved Model for DCT Coefficient Quantization*, Proceedings of the SPIE - The International Society for Optical Engineering,1993, Vol.1913, pp. 191-201

[59] C. Petzold, *Video for Windows brings interleaved audio and full-motion digital video to the PC*, Microsoft Systems Journal Vol. 8, Num.1, Jan, 1993, pp. 43-52.

[60] Pixie profiling software, Digital Equipment Corporation, 1991.

[61] T. Porter and T. Duff, *Compositing Digital Images,* SIGGRAPH '84 Proceedings, Vol. 18, pp. 253-259, July 1984

[62] W. Pratt, *Multimedia System Services, Version 1.0*, Interactive Multimedia

Association Compatibility Project. Technical Contact: Dr. William Pratt, SunSoft Inc., Mountain View, CA. william.pratt@sun.com

[63] K. R. Rao and P. Kip, *Discrete Cosine Transform -- Algorithms, Advantages, Applications*, Academic Press, Inc. London, 1990

[64] R. Riesz, E. Klemmer, *Subject Evaluation of Delay and Echo Suppressors in Telephone Communications*, The Bell System Technical Journal, Vol. 42, November, 1963, pp. 2919-2941.

[65] G. Rossum, et. al., *CMIFed: A Presentation Environment for Portable Hypermedia Documents*, Proceedings of ACM Multimedia 93, (Anaheim, CA, August 1-6, 1993). Association for Computing Machinery Press, New York, 1993 pp. 183-188

[66] L. A. Rowe, *personal communication*, 1994

[67] L. A. Rowe, K. Patel, B. C. Smith, *MPEG video in software: representation, transmission and playback*, IS&T/SPIE Symposium on Electronic Imaging: Science & Technology, San Jose, California, February, 1994.

[68] L. A. Rowe, B. C. Smith, *A Continuous Media Player*, Network and Operating System Support for Digital Audio and Video: Third International Workshop, (La Jolla, California, USA November 12-13, 1992). Springer-Verlag, Berlin; New York pp. 334-344

[69] *RTV Toolkit Software Guide, Version 2.2*, Parallax Graphics, Inc. Santa Clara, CA. 1993

[70] J. A. Saghri, et. al, *Image Quality Measure Based on a Human Visual System Model*, Optical Engineering, Vol. 28, Num.7, July 1989

[71] R. Sedgewick, *Algorithms*, 2nd ed., 1988, Addison-Wesley, Reading, Mass

[72] L. Shaw-Min, *Forward Error Correction Codes for MPEG2 over ATM*, IEEE

Transactions on Circuits and Systems for Video Technology, April 1994, Vol. 4, Num.2, pp. 200-203

[73] E. E. Smith, J. D. Lehman, *Interactive Video: Implications of the Literature for Science Education*, Journal of Computers in Mathematics and Science Teaching, Vol. 8, Num. 1, Fall, 1988, pp. 25-32

[74] R. Steinmetz, J. C. Fritzche, *Abstractions for Continuous Media Programming*, Computer Communications, Vol. 15, Num.6, July 1992, pp. 396-402

[75] W. Stevens, *TCP/IP Illustrated*, Addison-Wesley, New York, 1994.

[76] D. Stone, K. Jeffay, *Queue Monitoring: A Delay Jitter Management Policy*, Network and Operating System Support for Digital Audio and Video: Fourth International Workshop, Lancaster, UK November,1993. Springer-Verlag, Berlin; New York pp. 151-162

[77] F. Tobagi, J. Pang, *Starworks *TM -- A Video Applications Server*, Proceedings of IEEE Compcon '93 (San Francisco, California, USA, February, 1993).

[78] C. Topolic, *Experimental Internet Stream Protocol, Version 2 (ST-II); RFC-1190*, Internet Request for Comments, Num. 1190, Network Information Center, October 1990

[79] D. Verma, H. Zhang, *Design documents for RTIP/RMTP*, unpublished manuscript (1991).

[80] R. V. Volkman, *The digital video interface for Windows multimedia*, Windows-DOS Developer's Journal, Vol. 3, Num.9, Sept., 1992, pp. 5-14

[81] S. Voran, *The Development of Objective Video Quality Measures that Emulate Human Perception*, Proceedings of IEEE Global Telecommunications Conference: GLOBECOM '91, Phoenix, AZ, December 1991

[82] G. K. Wallace, *The JPEG Still Picture Compression Standard*, Communications of the ACM, Vol. 34, Num. 4, pp. 30-44, April 1991.

[83] A. B. Watson, *DCT Quantization Matrices Optimized for Individual Images*, Proceedings of the SPIE - The International Society for Optical Engineering,1993, Vol.1913, pp. 202-216

[84] A. Watson, *OMG (Object Management Group) architecture and CORBA (common object request broker architecture) specification*, IEE Colloquium on Distributed Object Management, Digest Num.1994/007, London, UK, Jan. 1994, pp. 4/1.

[85] R. Yavatkar, L. Manoj, *Optimistic Strategies for Large Scale Dissemination of Multimedia Information*, Proceedings of ACM Multimedia 93, (Anaheim, CA, August 1-6, 1993). Association for Computing Machinery Press, New York, 1993, pp. 13-20

[86] C. Zetzsche, *Principal Features of Human Vision in the Context of Image Quality Models*, Third International Conference on Image Processing and its Applications, Warwick, UK, July 1989.

[87] J. Ziv, A. Lempel, *A universal algorithm for sequential data compression* IEEE Transactions on Information Theory, May 1977, Vol. IT-23, Num. 3, pp. 337-343.

Appendix A
Geometric Transforms as LGDVEs

This appendix shows how to convert a geometric image transformation into a linear, global digital video effect (LGDVE). Such transformations include scaling, rotation, and shearing. We use the model of transformation plus filtering proposed in [23].

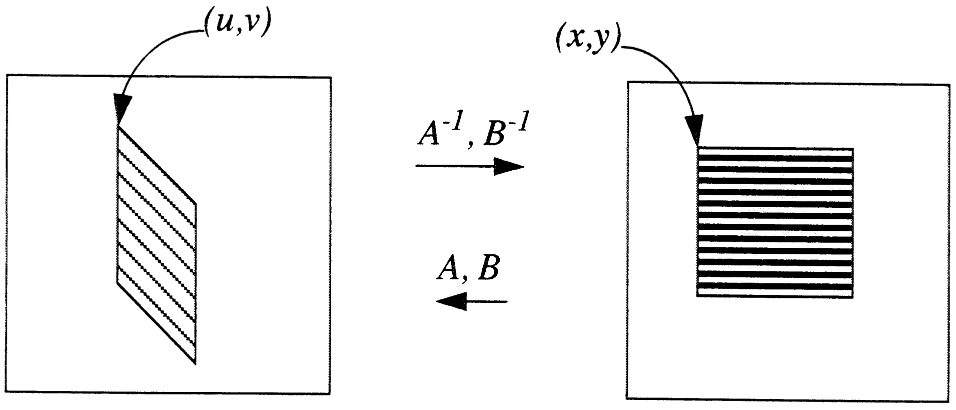
In this model, we are given a matrix geometric transform

$$\begin{aligned} u &= A(x, y) \qquad & x &= A^{-1}(u, v) \\ v &= B(x, y) \qquad & y &= B^{-1}(u, v) \end{aligned} \qquad \text{EQ A-1}$$

and filter function

$$g(x, y)$$

as shown in the figure below. Our goal is to compute the DVE transformation



tensor t_{uv}^{ij} from equation 3-1.

The value of an output pixel h_{uv} is given by

$$h_{uv} = \frac{1}{\kappa}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} g\left[A^{-1}(u, v) - x, B^{-1}(u, v) - y\right] f_{xy}\,dx\,dy \qquad \text{EQ A-2}$$

where κ is a normalization constant:

$$\kappa = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} g\left[A^{-1}(u, v) - x, B^{-1}(u, v) - y\right] dxdy \qquad \text{EQ A-3}$$

From the definition of the transformation tensor of a LGDVE given in equation 3-1, equation A-2 reveals that t_{uv}^{ij} is given by

$$t_{uv}^{ij} = \frac{1}{\kappa}\int_{0}^{1}\int_{0}^{1} g\left[A^{-1}(u, v) - (i+x), B^{-1}(u, v) - (j+y)\right] dxdy \qquad \text{EQ A-4}$$

which is the required tensor.