Maximizing QoS for Interactive DTV Clients

Edward Y. Chang
Department of Electrical Engineering
Stanford University
echang@cs.stanford.edu

Abstract

In this study we present a client-based architecture that supports interactive DTV features (i.e., pause, replay and fast-forward) using the client's local disk and an Internet back channel, together with the broadcast channels. Our study focuses on devising effective techniques for managing the client's local resources to maximize QoS. For individual video streams, we present an error concealment scheme that reconstructs video key frames suffering from channel errors. For the client as a whole, we propose a resource scheduling policy that maximizes the client's QoS adaptively based on the viewer's preference under local resource constraints. We show that many interactive DTV applications can be supported under this client-based architecture more effectively than by the traditional server-based approaches.
Keywords: digital TV, error concealment, disk scheduling, memory management.

1 Introduction

Many studies have proposed server-based interactive Digital TV (DTV) architectures (e.g., [3, 4, 6, 9, 15]), in which a server schedules its resources (i.e., memory and disk bandwidth) to service interactive VCR-like requests, such as pause, replay and fast-forward. In a server-based architecture, the clients are assumed to be passive—simply receiving bits and rendering frames. However, because of the typical long end-to-end transmission delay between a server and a client and the Interet's limited bandwidth, it is practically unfeasible for the server to support "real-time" interactive DTV features for tens of thousands of simultaneous users. Furthermore, on a broadcast channel (e.g., CNN), one simply cannot request the server to pause or replay a program.

In this study, we present a client-based interactive DTV architecture. A client in our architecture intelligently manages its local resources to support interactive DTV features. Equipped with a large and inexpensive disk, a DTV client can cache a large amount of media data [5]. This economical caching together with the random access capability of the disk enables a DTV client to support time-shift operations such as pause, instant replay and fast-forward. A viewer can pause a live TV program to take a break from viewing while the broadcast stream continues arriving and being written to the local disk. The viewer can resume watching the program after the pause with a delay, or fast-forward the program to get back in synch with the broadcast stream. These time-shift functions allow one, for example, to do one's own instant replay during a sports program or to watch a cooking lesson at one's own pace. A large cache also gives a client the flexibility to receive much more data than it consumes. For instance, video panoramas may deliver data to the DTV client at very high rates, but the client decodes, stitches, and displays only the frames that the user views. Without client assistance, these operations either cannot be implemented (in the broadcast case) or prolong playback time and hence may decrease server throughput (in the point-to-point transmission case).

Adding an Internet back-channel in addition to the disk to a DTV client allows a viewer to influence TV content in many ways. While a given broadcast stream remains the same for all viewers, various data objects can be transmitted to individual viewers. A data object can be textual (e.g., an HTML page), binary (e.g., a java appelet), or continuous (e.g., a video/audio stream). Image-based rendering techniques can overlay a number of data objects with the broadcast stream to provide a customized program. A viewer can query and change the attributes (e.g., color, shape, position, and history) of these data objects. A broadcaster can target individual families with tailored advertisement via the back-channel. Many other applications exist.

Managing heterogeneous DTV data objects (e.g., video, audio, texts, 3D graphics objects, etc.) at the client side to support interactive DTV features faces two major challenges: 1) each client may have different available resources and 2) each viewer may have different viewing preferences. A resource manager at the client side must allocate and schedule available resources adaptively to maximize each individual viewer's quality requirement. In this paper, we thus propose techniques to manage DTV data objects under the constraints of the local resources to maximize the client's QoS. We start with proposing a quantitative model that describes the size and latency characteristics for each data object. A viewer can assign a QoS factor to a data object to convey that object's scheduling priority to the resource manager. The resource manager then schedules the local resources for the data objects in an order that maximizes the total QoS. In addition, for real-time video data, we propose an error concealment scheme that maximizes QoS even when the data are delivered with errors. Through quantitative analysis, experiments and implementation we show that our client-based architecture can support interactive DTV features effectively with very low memory requirement and hence at a low cost. We believe that this client-based approach is a practical architecture to realize interactive DTV for both point-to-point and broadcast networks.

The rest of this paper is organized as follows: Section 2 depicts and analyzes a typical DTV pipeline, which consists of data objects and system resources. In Section 3 we sketch the resource allocation policies and derive memory use bounds. Section 4 describes how resources are scheduled to maximize QoS under constraints. Section 5 describes the packetization and image reconstruction scheme employed to improve image quality. Section 6 evaluates the effectiveness of our techniques. Finally, we offer our conclusion in Section 7.

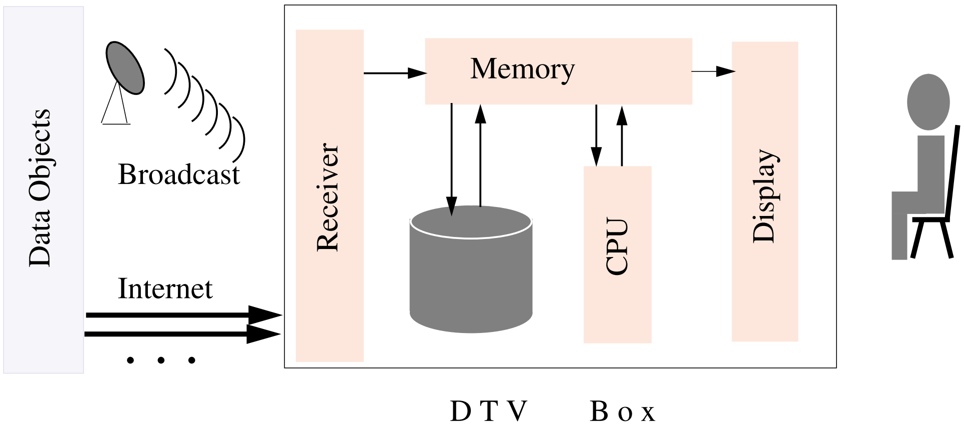


Figure 1: The DTV Pipeline

2 System Overview

Figure 1 depicts a DTV pipeline. On the left-hand side of the figure, DTV data objects are transmitted via broadcast and point-to-point channels to a DTV box. The DTV box receives network packets in its main memory. If the data are not needed shortly (e.g., the viewer paused the playback), the data are written to the box's local disk to conserve memory. The data are made available to the CPU before they are needed. The CPU processes (e.g., computes, decodes, renders, etc.) the data in main memory, and finally the processed data are played back to the viewer on the right-hand side of the figure.

A DTV pipeline consists of two major components: DTV data objects and system resources, including CPU, memory, and a local disk. (In this study we assume that a DTV box has only one local disk for economical reasons.) In the remainder of this section we analyze the characteristics of these two components and formulate quantitative models to represent their characteristics.

2.1 Characterizing Data Objects

DTV data objects can be continuous, textual, binary, and so forth. These data objects are best characterized by their *sizes* and *latency constraints*. Continuous data are audio/video streams, which are voluminous and delay sensitive. Textual data include captions, HTML and XML documents, etc. Textual data occupy far less space than continuous data and are not delay sensitive. Other data, such as images and applets, can be very copious or scant and may or may not be delay sensitive, depending on the nature of the application. Figure 2 shows the characteristics of these data objects. The x-axis in the figure represents the delay sensitivity from low (on the left-hand side) to high (on the right-hand side); the y-axis gives the size of an object. At one extreme, videos are large and very sensitive to delay. At the other extreme, captions are small in size and not very sensitive to delay.

The size of a data object is measured by the amount of storage space it needs. We denote the storage requirement for object i as S_i. In Section 3.2 we show how S_i is computed. To

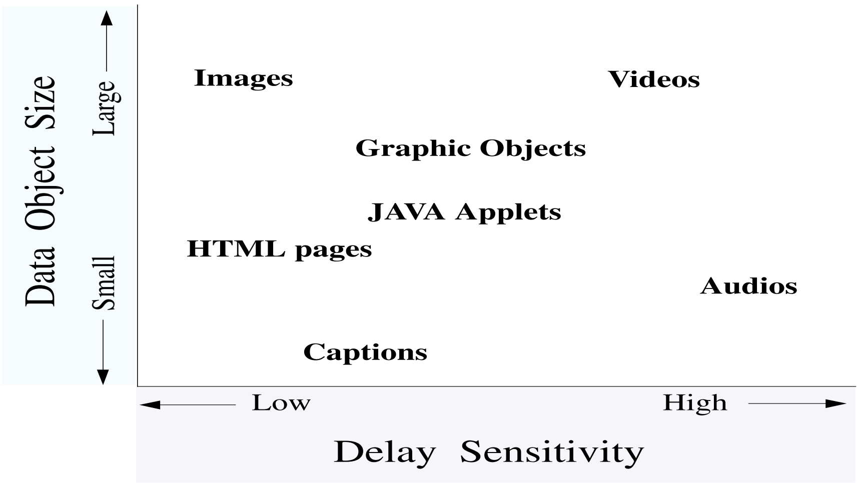


Figure 2: The Characteristics of DTV Data Objects

quantify the delay sensitivity of a data object, one can specify a value function for each data object. The function can be defined in many ways: Figure 3 shows four representatives. The x-axis in the figure depicts delay; the y-axis depicts the value of the object normalized to from zero (worthless) to one. Function F_a depicts an object that can tolerate delay up to a period of time, t_δ, then becomes worthless. Function F_b shows the value of an object that decays linearly after t_δ. Functions F_c and F_d have other value decay patterns. Function F_a is a good value function for an audio frame. Function F_b may be a good value function for a video frame since slight delay of a frame display may be tolerable. Without losing generality, we define the value function for object i as $f_i(t)$, where t denotes the span between the time the object is requested and the time the object is available in main memory for processing.

In short, we characteristize DTV data objects as follows:

- S_i represents the i^{th} object's storage requirement, and
- $f_i(t)$ depicts the i^{th} object's delay sensitivity.

2.2 Characterizing System Resources

The critical resources of a DTV box include CPU, memory, and a local disk. In this study, we focus on memory space and disk bandwidth management. These resources can also be characteristized by their capacity and latency. The memory size (denoted as M_{avail}) and the disk size (denoted as D_{avail}) are both constants that can be obtained from the system configuration file. The latency of a DTV box is best characteristized by the IO time between memory and the local disk since this latency dominates the latency in the storage hierarchy. The IO time for reading/writing data object i (denoted as τ_i) can be written as:

$$\tau_i = \gamma(d) + S_i/TR, \tag{1}$$

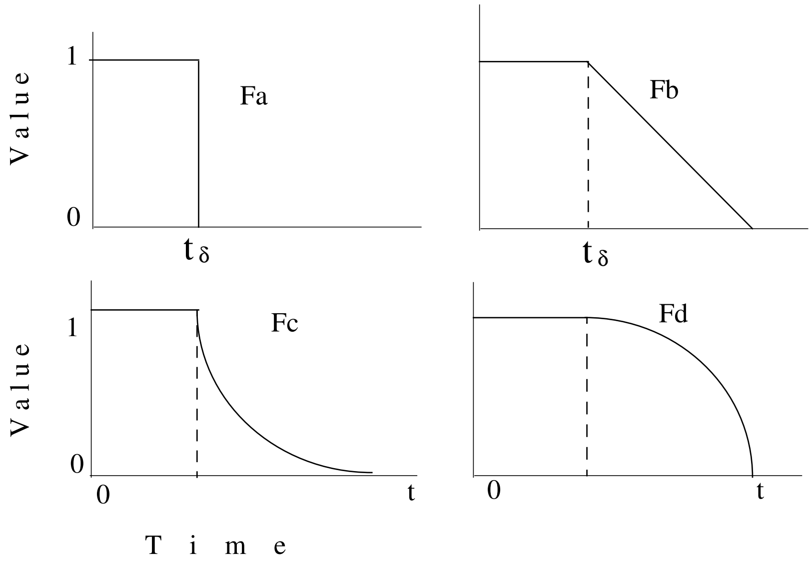


Figure 3: Delay Functions

where $\gamma(d)$ is the seek time and rotational delay of the disk given d tracks to seek before transferring data, S_i is the size of the i^{th} data object, and TR is the transfer rate of the disk. Note that the latency computed in Equation 1 assumes that no IO conflict exists. In reality, the more objects are accessed, the longer the delay that can be experienced by an IO. We take scheduling conflicts into consideration in Section 4 and devise methods that resolve the conflicts optimally with respect to the viewer's preference.

To assist the reader, Table 1 summarizes the parameters we have described, together with other parameters we will introduce later. The first tier of Table 1 lists the physical and derived characteristics of the hardware resources. The second tier shows the parameters and functions that depict the data supply and input rates. Finally, the third tier presents parameters that are used to schedule data objects.

3 Resource Allocation

In this section we show how the memory and disk bandwidth are allocated by a DTV client. We first categorize a DTV data object as either continuous or transient. Continuous data objects require continuous memory commitment. For transient data objects memory is allocated on demand only before and while the objects are being processed. Continuous data objects are audio/video streams and those data objects that must be played back in synch with the streams, e.g., captions and commercials. Transient objects are typically transmitted and processed as a result of the user's interactive request. Transient objects are therefore temporary and do not need to be stored on disk (i.e., involve no IOs).

The methods for estimating the memory requirement for continuous and transient data objects are different. For a transient data object, the memory requirement is independent of

Parameter	*Description*
M_{avail}	Total available memory space, in pages
M_c	Available memory space for continuous data, in pages
M_n	Available memory space for non-continuous data, in pages
p_{size}	Page size
S_i	Memory required by the i^{th} data object
D_{avail}	Available disk space, in blocks
TR	Disk transfer rate, in Mbps
$\gamma(d)$	Function that computes worst disk latency
τ_i	Worst IO latency for the i^{th} data object
T	Time an IO is guaranteed to be completed after issued
δ	Frame period
tt	Playback time of a video, in frame periods
x_i	The media data encoding function
y_i	The media data supply function
L	No. of data objects requested
α_i	QoS assigned to the i^{th} data object
N	No. of data objects selected to be scheduled
N_c	No. of continuous objects to be scheduled
N_n	No. of non-continuous objects to be scheduled

Table 1: Parameters

the concurrent presence of other objects. For instance, a 3D graphics object that requires 4 MBytes of memory to be rendered requires that same amount of memory regardless of what other data objects are processed at the same time. The memory requirement of a continuous data object, on the contrary, depends on how many IOs can concurrently be requested in the system and on the latency and transfer time of these IOs (e.g., [4]).

The resource manager separates the local memory into two pools, one for continuous and one for transient data. We assume that the transient memory requirement, denoted by M_t, is given. We analyze how the memory requirement for the continuous data, denoted by M_c, is computed. The total memory M_{avail} should be larger than or equal to the sum of the two, or $M_{avail} \geq M_c + M_t$.

3.1 Memory Use of Continuous Data

Figure 4 depicts a continuous media data delivery process. First, the encoded media data arrive at the *receiver* (on the left-hand side of the figure). The encoded bit stream can either be at a variable bitrate (VBR) or at a constant bitrate (CBR). We define the encoding function (and thus the decoding function in Figure 4) as $x_{i,j}$, where j denotes the j^{th} frame period, and $x_{i,j}$ the number of bits used to encode the j^{th} frame by the i^{th} object. For instance, a movie that lasts 100 minutes, with 30 frames per second, consists of $100 \times 60 \times 30 = 180,000$ frame periods, each lasting 33 milliseconds. The points in Figure 5 show an encoding function for an MPEG2 movie. Since the number of bits required for encoding the anchor frames (i.e., the I and P frames) is larger than that for the intercoded frames (i.e., the B frames), we see that $x_{i,j}$ fluctuates from period to period.

The *decoder* (on the right-hand side of Figure 4) uses an "inverse" scheme for playback, so

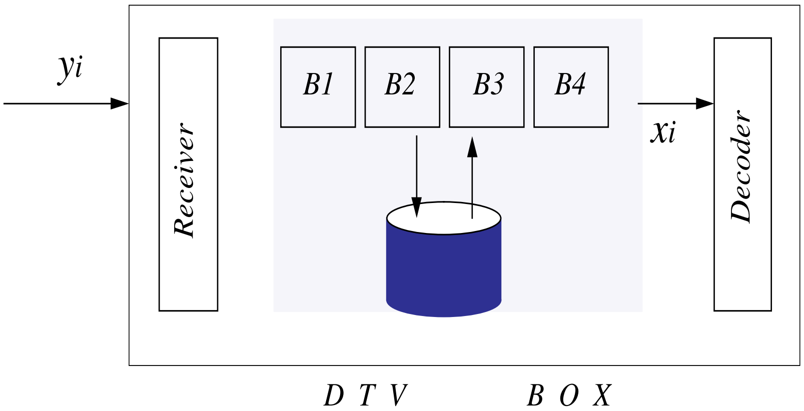


Figure 4: Four-buffer Ping-pong Scheme

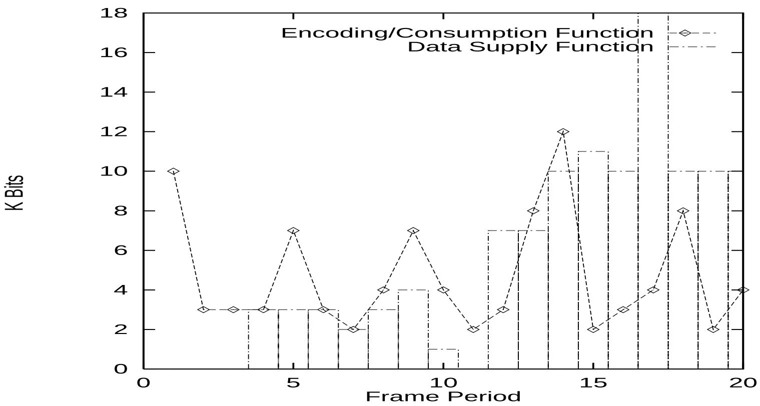


Figure 5: Functions

$x_{i,j}$ is also the data consumption function. (The decoder can infer how much data is needed at each point of time from the stream itself. For detailed specifications of coding standards like H.261 or H.263 consult references such as [1].)

Data can be delivered to the client with a delivery profile $y_{i,j}$ that may differ significantly from $x_{i,j}$. (The amount of data arriving for the i^{th} object in frame period j is given by $y_{i,j}$.) For example, data can be packetized and delivered to the client with a constant size and rate packetization (CSRP) scheme, a variable size and rate packetization (VSRP) scheme [8], or a combined scheme (e.g., constant size plus variable rate or variable size plus constant rate). Furthermore, depending on the channel capacity, the data delivery rate can be faster or slower than the data consumption rate. Finally, errors may also affect the delivery profile. The rectangles in Figure 5 show a sample $y_{i,j}$ function.

To cushion the difference between the input and playback functions, which can be very large (e.g., every one minute pause of a 19.2 Mbps DTV program requires 1.44 GBytes of buffer space), the client uses a memory and disk integrated cache, shown in the middle of Figure 4, to reduce memory requirement. For each continuous data object, the resource manager allocates four memory buffers of equal size (typically less than four MBytes). These four buffers are ping-ponged between the receiver and the decoder to be used to receive incoming data from the network, write data to the disk, read data from the disk, and provide data for decoding. As we will show shortly, to reduce memory requirement, the receiver and the decoder each "owns" at most two buffers at any time. The following example illustrates how the resource manager manages these four buffers for a video playback.

At the start of the video playback, the resource manager allocates four buffers, denoted as B_1, B_2, B_3, and B_4 (shown in Figure 4), for a stream. Once data arrives, the resource manager receives the data in one of the buffers, say B_1. When B_1 is full, B_1 is available for decoding and hence the resource manager hands B_1 to the decoder. (In this example, we assume that the playback starts once B_1 is filled up and is assigned to the decoder. The playback can start earlier or later, depending on the difference between the data delivery and playback rates. For example, if the data delivery rate is substantially slower than the playback rate, the resource manager must accumulate enough data on disk before the playback can start to prevent display glitches.) At the same time, the resource manager assigns another buffer, say B_2, to the receiver to continue receiving data. When buffer B_2 is full, the resource manager hands B_2 to the decoder and assigns B_3 to the receiver to continue receiving data. At this time, the decoder "owns" buffers B_1 and B_2, and the receiver B_3.

One of two events occurs next: either the data in buffer B_1 is used up by the decoder or buffer B_3 is filled up by the incoming packets. The resource manager takes different actions according to which event occurs first.

- If B_1 is used up first, the resource manager simply returns the buffer to the free pool.
- If B_3 is filled up first, the resource manager writes the buffer to the disk. Note that the decoder does not need B_3 immediately. The decoder needs only B_2 to safeguard a glitch-free playback. When B_3 is being written to the disk, B_1 and B_2 are held by the decoder. The resource manager must use buffer B_4 to receive incoming data. B_4 must be large enough for the disk to complete writing B_3 to disk, or the receiver runs out of buffer to receive data once B_4 is full.

At this time, one of two things can happen next: the data in buffer B_1 is used up by the decoder or buffer B_4 is filled up by the arriving data.

- When B_1 has been consumed by the decoder, the resource manager starts consuming B_2. At the same time, the resource manager has to make sure that the data after that in B_2 is made available to the decoder before the decoder uses up B_2, or hiccups occur. If the decoder uses up B_1 before B_3 is full, the resource manager assigns B_3 to the decoder as soon as the buffer is filled up. If B_3 is filled up before buffer B_1 is consumed, an IO to write B_3 to the disk may be in progress or may have been completed. In the former case, the resource manager still makes B_3 available to the decoder by ignoring the write (canceling the write may not be possible). If the write has been completed, the resource manager reads the data back from the disk into B_1 (the decoder uses the data in B_1 after

B_2 is consumed). Note that the size of buffer B_2 must be large enough to allow enough time for buffer B_1 to be replenished.

- If buffer B_4 is filled up first, the resource manager writes the data to the local disk. Since either buffer B_1 or B_3 must have been free at this time, the resource manager uses the available buffer as the receiving buffer.

In summary, the resource manager is driven by two events: *buffer-consumed*, when the decoder has consumed a buffer, and *buffer-full*, when the receiver has filled up a buffer. When the *buffer-consumed* event occurs, the resource manager may need to read a buffer of data from the disk before the decoder uses up the next buffer. The buffer thus must be large enough to provide data to the decoder before the read is completed. When the *buffer-full* event occurs, the resource manager may need to write the buffer to the disk to free up space for use. The buffer must be large enough so that during the write the receiver cannot fill up another buffer. This four-buffer ping-pong scheme enjoys three benefits:

1. It serves as a rubber-band to cushion the difference between the incoming and decoding bitrates.
2. It does not perform unnecessary memory-to-memory or memory-to-disk copy if the data is played back in real-time.
3. If the delay is desirable (a viewer pauses the playback), it uses the disk instead of memory to buffer data and limits the memory requirement to four buffers. This ping-pong scheme minimizes memory requirement and hence cost to support time-shift operations. (In Section 6.1 we show that the typical memory requirement is less than four MBytes for a 19.2 Mbps video stream.)

3.2 Quantitative Analysis

In this section we analyze the client's use of main memory. Our design goal is to avoid, as much as possible, display glitches due to variability and delays in the input stream. Studies have shown that even just losing 0.1% of data can cause significant degradation in display quality, resulting from inter-frame decoding dependencies [7, 8]. Therefore, in computing our control parameters we take a conservative approach, that is, we assume the worst-case disk latency, as well as peak data consumption and input rates.

Let $T1_i$ denote the longest time that it takes to complete a read for the i^{th} continuous data object. The maximum amount of data that the decoder can consume in $T1_i$, denoted as $Max_{x_i}(T1_i)$, can be expressed as

$$Max_{x_i}(T1_i) = \max_{0 \le \tau \le tt} \sum_{j=\tau}^{\tau + \lceil \frac{T1_i}{\delta} \rceil - 1} x_{i,j},$$

where tt denotes the total number of frame periods in x_i. Note that the buffer size must be at least $Max_{x_i}(T1_i)$ or we may run out of data for the decoder during the $T1_i$ period, causing a glitch. Let p_{size} denote the page size. In terms of the number of pages, we can express the minimum buffer size for the i^{th} data object $\sigma 1_i$ as

$$\sigma 1_i \geq \lceil \frac{Max_{x_i}(T1_i)}{p_{size}} \rceil. \quad (2)$$

To conserve memory, we take the equality in the above expression, yielding

$$\sigma 1_i = \lceil \frac{Max_{x_i}(T1_i)}{p_{size}} \rceil. \quad (3)$$

Similarly, let $T2_i$ denote the longest time the disk takes to complete writing out a buffer of data for the i^{th} continuous data object. The maximum amount of data that can possibly arrive in $T2_i$ time at the client can be expressed as

$$Max_{y_i}(T2_i) = \max_{0 \leq \tau \leq tt} \sum_{j=\tau}^{\tau + \lceil \frac{T2_i}{\delta} \rceil - 1} y_{i,j}.$$

In terms of pages, the minimum buffer size, $\sigma 2_i$, is given by

$$\sigma 2_i = \lceil \frac{Max_{y_i}(T2_i)}{p_{size}} \rceil. \quad (4)$$

The buffer selected must be the larger of the two, or $Max\ \{\sigma 1_i, \sigma 2_i\}$. Since each continuous data object requires four buffers, the memory requirement for the i^{th} continuous data object is

$$S_i = 4 \times Max\ \{\sigma 1_i, \sigma 2_i\}. \quad (5)$$

Next, we derive the formulas to compute $T1_i$ and $T2_i$ so that we can solve for S_i. Since reads and writes can interleave, a read request may not be scheduled until a write completes and vice versa. In addition, there may be more than one continuous data object arriving at and being played back by the client. Both $T1_i$ and $T2_i$ therefore must account for the worst-case delay before a read and write request can be completed.

Assume we have N_c continuous data objects that request IOs. Assume that a round of IO time for the continuous data objects is

$$T = N_c \times \gamma(d) + \sum_{i=1}^{N_c} S_i \times p_{size}/TR.$$

Finally, setting $T1_i$ and $T2_i$ equal to T guarantees two things: the arriving data will not overflow the buffer and the data supply to the decoder will not underflow for the i^{th} object. Given x_i, y_i, the page size and disk parameters, we can solve for S_i.

4 Scheduling Resources to Maximize QoS

So far, we are given the characteristics of the data objects to compute the resource requirement. However, the local resources may not be adequate to support a particular interactive scenario. Therefore, it is critical for the resource manager to be adaptive to the resource constraints and to degrade, if degradation is unavoidable, in a graceful manner. Given limited memory and disk bandwidth, the design objective of a DTV box is to prioritize resource allocation in order to maximize the user's satisfaction.

4.1 Maximizing QoS

To measure a user's satisfaction, we must allow the user to have a say about what is important and what is not. We thus associate each data object with a *QoS parameter*. A viewer can assign a QoS value to each data object, either implicitly or explicitly. For instance, if a user decides to turn off all interactive features, the resource manager can assign a QoS factor of zero to data objects that are needed to support interactive features. Regarding the data objects that are needed to support the playback, higher QoS can be assigned to mission-critical data objects, such as broadcast video and audio streams and lower QoS can be assigned to optional data objects such as captions and applets.

Given N_{all} requested data objects, the goal of the resource manager is to schedule N data objects ($N \leq N_{all}$) to maximize the total QoS under the resource constraint. We use α_i to denote the QoS requirement assigned to the i^{th} data object ($0 \leq \alpha_i \leq 1$). Let j represent the scheduling order for N data objects. We would like to schedule the service to these N data objects in an order that maximizes the sum of the QoS. The objective function can be written as:

$$Max \sum_{j=1}^{N} \alpha_j \times f_j(t - \sum_{k=1}^{j} \tau_k), \tag{6}$$

which is subject to the memory constraint

$$\sum_{j=1}^{N} S_j \leq M_{avail}.$$

The following example illustrates how the resource manager schedules work under resource constraints.

Suppose a DTV box needs to process three data objects: O_1, O_2, and O_3. Suppose these objects have the characteristics depicted in Table 2 and the DTV box has $M_{avail} = 4$ MBytes of DRAM. To simplify our discussion, we precompute the latency τ_i (the IO time to transfer S_i amount of data) for each object in the table. Suppose each frame period is 33 ms. The latency functions $f_a(t)$ and $f_b(t)$ are depicted in Figure 6.

With the 4 MBytes DRAM constraint, the resource manager cannot process all three data objects since doing so requires 4.205 MBytes of DRAM. The resource manager can however, process any two objects in any orders, and it thus has six feasible schedules: $\{O_1, O_2\}$, $\{O_2, O_1\}$, $\{O_1, O_3\}$, $\{O_3, O_1\}$, $\{O_2, O_3\}$, and $\{O_3, O_2\}$.

O_i	S_i	$f_i(t)$	α_i	τ_i
O_1	3 MBytes	$f_b(t)$	1	20 ms
O_2	5 KBytes	$f_a(t)$	1	15 ms
O_3	1.2 MBytes	$f_b(t)$	0.3	18 ms

Table 2: Example Data Objects and their Characteristics

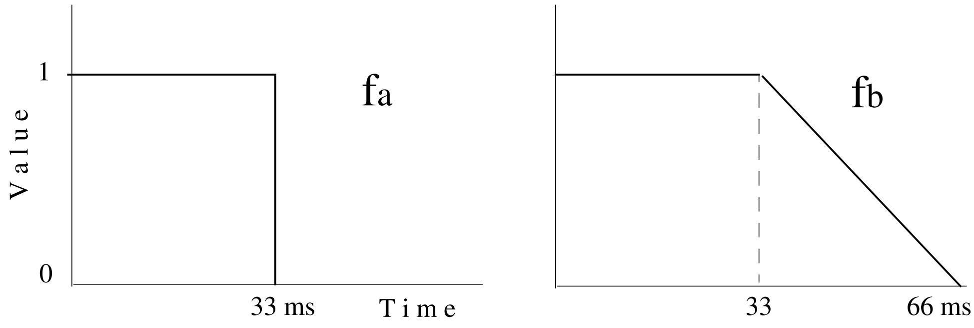


Figure 6: Latency Functions $f_a(t)$ and $f_b(t)$

To compute the total quality that a schedule can achieve, we plug all given statistics in Table 2 into Equation 6. For instance, for the first schedule $\{O_1, O_2\}$, the resource manager performs an IO to read data for O_1 then O_2. The latency to perform IO for O_1 is 20 ms and the latency function $f_b(t-20)$ returns one. However, when the IO is completed for O_2, the latency function $f_a(t-35)$ returns zero. (The excessive delay of retrieving O_2 makes the object worthless to the viewer.) The total QoS is thus $1 \times 1 + 1 \times 0 = 1$. If we switch the service order to O_2 then O_1, the latency for reading O_2 is 15 ms and the latency function $f_a(t-15)$ returns one. When the IO is completed for O_1, the latency function $f_b(t-35)$ returns 0.94. The total QoS is 1.94, a better total QoS than we get by processing O_1 first and then O_2. Table 3 lists the total QoSs for all six feasible schedules. The schedule that achieves the maximum QoS is $\{O_2, O_1\}$. In other words, under the resource constraint, retrieving first O_2 and then O_1 achieves the maximum QoS.

Note that different clients may receive the same data objects, but based on the QoS factors they assign to these objects and the local resource constraints, the resource manager allocates and schedules resources adaptively to achieve different quality results. In the example above, if we assign to the audio object a QoS factor of 0.3 and to the image a QoS factor of 1.0, the optimal schedule becomes $\{O_1, O_3\}$. Similarly, changing the latency functions and the resource constraints also alters the optimal schedule. Therefore, the resource management scheme we propose is adaptive to user preferences, data characteristics, and resource constraints.

4.2 Solving the Optimal Schedule

A brute-force method to solve the optimal schedule is to enumerate all possible permutations of N_{all} requests and select the permutation that produces the maximum total QoS. This approach, however, can be very expensive if N_{all} is large. For example, when $N_{all} = 5$, a probable scenario,

Schedule	τ_1	$\tau_1 + \tau_2$	*Total QoS*
$\{O_1, O_2\}$	20 ms	35 ms	1
$\{O_2, O_1\}$	15 ms	35 ms	1.94
$\{O_1, O_3\}$	20 ms	38 ms	1.28
$\{O_3, O_1\}$	18 ms	38 ms	1.15
$\{O_2, O_3\}$	15 ms	33 ms	1.3
$\{O_3, O_2\}$	18 ms	33 ms	1.3

Table 3: QoSs of Feasible Schedules

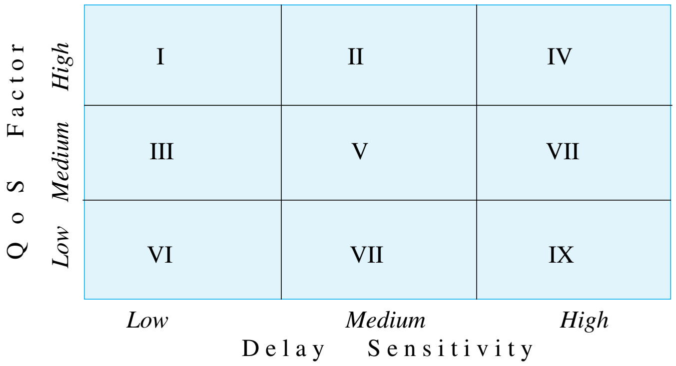


Figure 7: IO Scheduling Table

the total number of permutations is 120 (roughly on the order of $O(N_{all}^2)$). Since we do not want the computation of the optimal schedule itself to become a delay factor, we use an efficient method that has a linear cost (i.e., the cost of computation is on the order of $O(N_{all})$) to approximate the result.

To reduce the computation cost, we put the α values into three categories: low, medium, and high. We also put the delay sensitivity of an object into these three categories. We then plot N_{all} data objects in the schedule table as shown in Figure 7. The x-axis of the figure is the delay sensitivity and the y-axis the QoS factor. After N_{all} IO requests find their quadrants in the schedule table, the resource manager schedules the IOs from quadrant I to IX in ascending order. This method effectively separates the highest and the lowest priority tasks from the rest and the resource manager schedules the highest priority tasks first and abandons the lowest ones if necessary.

5 Maximizing QoS Under Errors

We have this far assumed that the data delivery is error free. However, due to channel errors, signals can be corrupted. To address the problem of the lack of resiliency to channel errors, layered coding and error correction schemes such as automatic retransmission query protocols (ARQ) or forward error correction (FEC) have been proposed [13]. However, if the network is already congested, ARQ tends to aggravate the problem by sending out retransmitted packets.

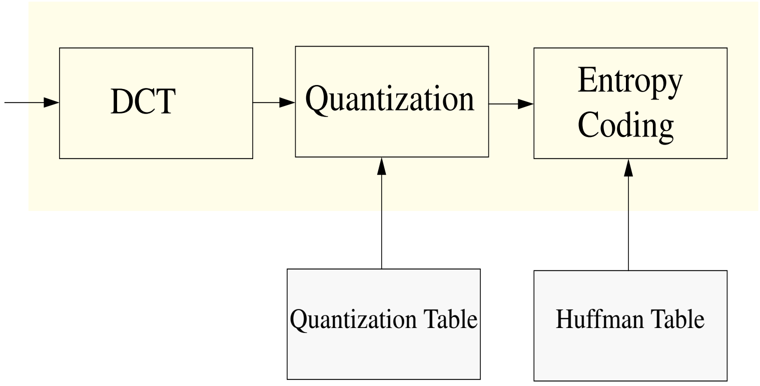


Figure 8: JPEG Encoding Procedure

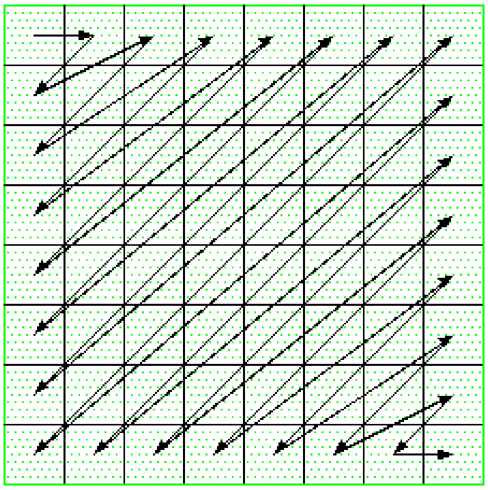
Also, retransmission does not help real-time decoding schemes such as MPEG. As for error correction codes, they are effective only when the loss rate is below the design threshold. Furthermore, the redundant transmissions and codes required to improve reliability decrease the effective channel capacity. For example, some low bit rate, high redundancy coding schemes (e.g., ARQ-I and ARQ-II [18]) allocate bits for FEC. This pessimistically degrades image quality (i.e., lower SNRs) for errors that may not occur.

This section presents research on an image packetization and reconstruction scheme that works for the key frames (i.e., I frames) of MPEG video streams. It is critical to repair a damaged key frame, or otherwise the error may propogate to the inter-coded frames, whose decoding depends on the key frames. (If a inter-coded frame is damaged, we simply disregard the damage since the error is well contained.) Specifically, our design objectives have aimed at 1) preventing propagation of errors, 2) smoothing out the effects of bursty loss, and 3) reconstructing damaged frames accurately and efficiently. The study is based on the existing JPEG international standards. (We conducted our initial experiments on JPEG images in [2].) However, since standards such as H.261 and H.263 use compression schemes similar to JPEG to perform intra-frame encoding, the techniques that work for JPEG also work for video key frames.

5.1 A Brief Overview of JPEG

To assist understanding of the techniques proposed in this section, we give a brief overview of JPEG [17]. The JPEG encoding procedure consists of three steps: DCT (Discrete Cosine Transform), quantization, and entropy coding. JPEG decoding, conversely, consists of entropy decoding, dequantization, and IDCT (Inverse Discrete Cosine Transform). Figure 8 depicts the encoding steps; the decoding steps are executed in the opposite direction. The following describes the encoding steps only.

JPEG deals with colors in the YUV (one luminance and two chrominances) space. For each separate color component, the image is broken into 8 by 8 pixel blocks of picture elements, which join end to end across the image. JPEG transforms each block into a two-dimensional DCT matrix by performing a one-dimensional DCT on the columns and on the rows. Figure 9(a) shows an 8 by 8 coefficient matrix generated by the DCT step. The arrows link the frequency components from DC to the highest. Note that at this point the information about the original image is preserved with the exception of that lost due to the rounding errors of the DCT coefficients.



(a) DCT Coefficient Matrix

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

(b) Quantization Table

Figure 9: DCT Coefficients

The coefficients of the DCT are quantized to reduce their magnitude, to increase the number of zero value coefficients, and to reduce the bit rate. Figure 9(b) shows the uniform midstep quantizer that is used for the JPEG baseline method, where the step size varies according to the coefficient location in the DCT matrix. As one can see, the low frequency components (at the upper left corner of the quantization matrix) have smaller steps, while the high ones have larger steps. This is because human eyes are less sensitive to high frequency components. Quantization is the lossy stage in the JPEG coding scheme. If we quantize too coarsely, we achieve a higher compression rate with poorer image quality. On the other hand, if we quantize too finely, we may spend extra bits on coding noise.

The final step is entropy coding. JPEG uses the Huffman or arithmetic coding scheme to compress the quantized DCT coefficients to approach the entropy rate. Since both the Huffman and the arithmetic codes are variable-rate, any bit errors or loss in the transmission propagates and destroys the entire image. One remedy to this problem is to introduce resynchronization points in the code to limit the effect of the loss. Another way is to use fixed-rate coding schemes [14] that trade bit rates for error containment. With minimum changes to the current JPEG baseline, we use resynchronization bits to contain the loss effect. The proposed packetization and reconstruction schemes work orthogonally with variable or fixed-rate codes.

5.2 Packetization and Reconstruction

Our proposed scheme to combat contiguous bit loss consists of two parts: packetization and reconstruction. The objective of the packetization step is to scatter the bursty loss in the spatial domain so that adjacent DCT blocks do not lose the same frequency components. This aids the reconstruction quality since 1) a block does not lose all its coefficients and 2) most lost frequency components can be recovered from adjacent blocks. In the reconstruction step, we also take spatial characteristics of the frequency components into consideration to repair the damaged image more accurately.

5.2.1 Packetization

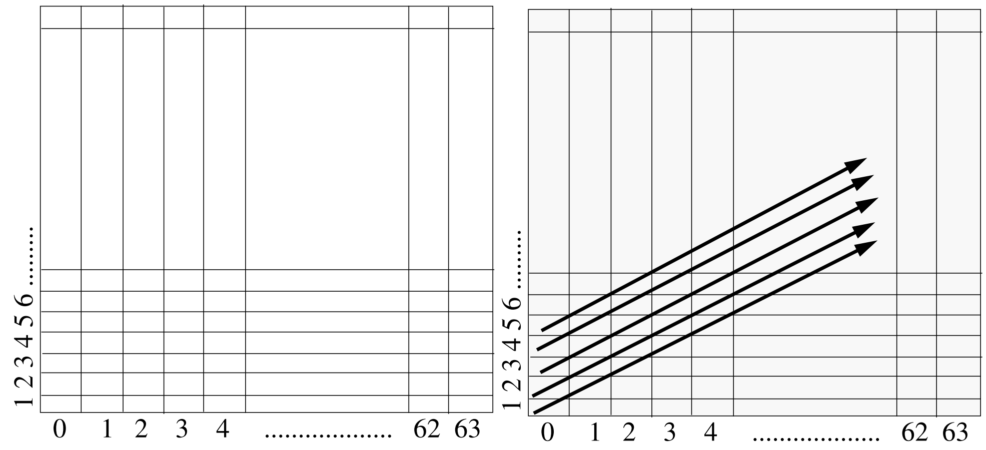
To minimize the effect of bursty packet loss on an image, the packetization scheme must achieve two goals in the spatial domain. First, for each burst of error, the lost information must not be clustered on the image. Second, the spatially adjacent 8 by 8 blocks of the image must not lose the same frequency components. If both objectives are met, each block will lose only a small number of coefficients, and the lost frequency components will have a high chance of being recovered from the neighboring blocks (which do not lose the same frequency components).

Without loss of generality, one can assume that each frame is divided into an H (horizontal) times V (vertical) number of 8 by 8 blocks. Designating the code of each block as C(h,v), one can construct the H times V code array shown in Figure 10(a). The horizontal axis in Figure 10(a) represents the frequency components of each block from 0 to 63. The vertical axis of the figure represents the blocks ordered by their spatial locations from the top left-hand corner (h = 0 and v = 0) of the image to the bottom right-hand corner (h = H and v = V).

To make sure that the lost coefficients are shared by as many blocks as possible, each transmission unit (the new block constructed from the DCT blocks) includes frequency components from 64 blocks. Figure 10(b) illustrates a simple way to achieve this objective by packing frequency components diagonally. Note that entropy coding is performed after this packetization step, and hence the packetization scheme is not affected by whether the resulting code is fixed or variable-rate.

Packing coefficients from different blocks achieves only part of the objective. Next, one wants to ensure that the spatially adjacent blocks do not lose the same frequency components when packets are dropped. Let T (top), B (bottom), L (left), and R (right) denote the spatially adjacent DCT blocks, as depicted in Figure 11. Section 4.2 will show that the lost coefficients can be reconstructed from a block's four neighboring blocks (except for the blocks on the edges). However, to ensure that the neighboring blocks do not lose the same frequency components, i.e., to optimize the reconstruction, we propose packetizing these transmission units in "strides."

A stride is the number of blocks skipped between packing blocks. For instance, if the stride is 2, we packetize blocks in the order of 1, 3, 5, 7, . . . and then 2, 4, 6, 8, . . ., etc. This way, for example, if blocks 3, 5 and 7 are lost in a burst, the DC components of the lost blocks can be reconstructed from blocks 2 and 4 for 3, 4 and 6 for 5, and so forth in the horizontal direction of the image. To make sure that in the vertical direction no spatially adjacent blocks



(a) DCT Coefficient Array *(b) Block Packetization*

Figure 10: DCT Coefficient Packetization

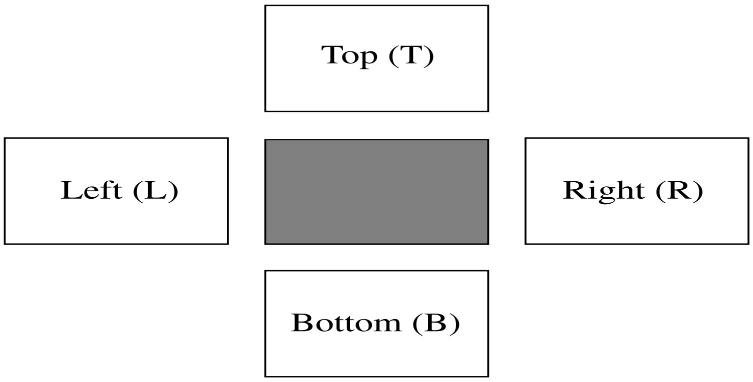


Figure 11: Spatially Adjacent Blocks

lose the same coefficients, one must shift the starting point of the stride from row to row. For instance, one can let the stride on the odd rows start from the first image column while that on the even rows starts from the second column. This way, even if the contiguous loss is as high as 50%, the neighbors of a lost block are still intact for reconstruction. In an environment where the loss rate is much lower, one can choose a larger stride to disperse the artifacts as long as the stride is prime to the width (in blocks) of the image.

To summarize, the coding scheme first packetizes the blocks diagonally to spread the frequency components to 64 transmission units. Then, it uses a stride method to pack these transmission units in such a way as to turn the potential consecutive block loss into a pseudo-random loss. To invert the packetization, the only information the decoder needs is the stride. This number can be sent with every packet with a negligible overhead (less than 4 bits) compared to the size of a compressed image.

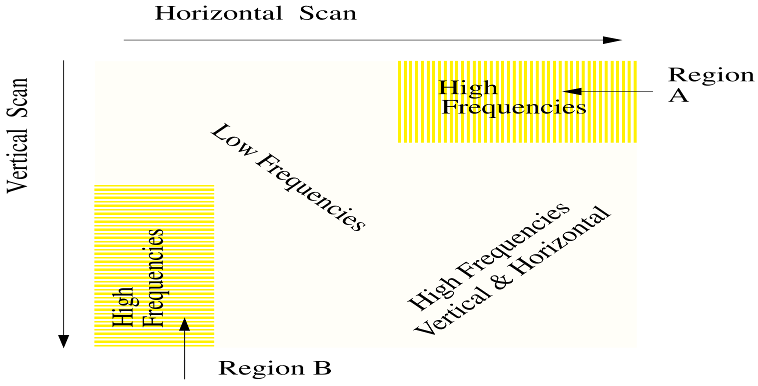


Figure 12: Frequency Component Characteristics

5.2.2 Reconstruction

The studies of [10, 11] show two typical approaches to reconstruct a damaged image. The first approach, introduced in [11] is a decoder side reconstruction that interpolates the T, B, L, and R blocks to reconstruct the lost block. The technique used by [11] adds only 20% in computational overhead to the decoder, and the reconstruction is of higher quality than that achieved in the previous attempts. The reconstruction formula can be depicted as follow:

$$C_z = C_z + W_t \times C_t + W_b \times C_b + W_l \times C_l + W_r \times C_r \tag{7}$$

where the weights, W_t, W_b, W_l, and W_r, are the weighting factors for averaging the neighboring blocks' coefficients to reconstruct coefficient C_z.

In a later study [10], Hemami proposes an encoder side approach in which the "optimal" weight vectors are computed for each DCT block based on 15 different combinations of available adjacent blocks (none available, one available, etc.). During the reconstruction step, for each block lost, the proper weighted vector is selected to perform linear interpolation. These vectors, since they take up large space ($200 - 600\%$ of the image data), are compressed using a vector quantizer before the image is shipped to the clients. The resulting space overhead, as reported, is reduced to about 10% of the image size. The drawback of this approach is that either the disk space requirement at the server side increases by 110% (to keep two different compressed images), or the bit rate decreases by 10% (the inflated version is transmitted regardless of the channel's characteristics). Another concern is that this approach greatly increases encoding time. Moreover, the scheme, although achieving better reconstruction quality, may not be optimal after all. We argue that the optimal weight set must be at the frequency component level rather than the block level. Since the optimal weight set is computationally intensive and takes up large space, expecting an optimal reconstruction may not be practical. Consequently, we suggest a simple and intuitive method to set the weight factors.

The following presents a much simpler heuristic approach that incurs no extra bit rate or computational overhead for image reconstruction. Figure 12 shows the 8 by 8 DCT coefficient matrix by its spatial characteristics. The high-frequency components corresponding to the horizontal DCT scan (region A in the figure) represent the vertical edges in the image (e.g., a tree), and the high frequency components corresponding to the vertical DCT scan (region B in the figure) represent the horizontal edges (e.g., a roof top). To take advantage of this relationship, the interpolation function (Eq. 7) should assign different weights for different frequency components. The higher the frequencies in the vertical scan (representing horizontal lines), the smaller the weight one should assign to the top and bottom blocks. On the other hand, the higher the frequencies in the horizontal scan (representing vertical lines), the smaller the weight one should assign to the left and right blocks. For example, for reconstructing a vertical line (e.g., a tree), only the top (T) and bottom (B) blocks of the missing block are relevant. The left (L) and right (R) blocks of the missing block do not have the same tree! If one added left and right blocks into the computation for reconstructing the vertical line, one would dilute the recovered coefficients. Of course, one cannot know whether a DTC block contains a tree or not. However, if a tree passes through the missing block, the high frequencies of the horizontal scan must not be zero, and this value can be recovered from the blocks above and below. On the other hand, if no tree passes through the missing block (in other words if the high frequency components in the horizontal scan are zero), there is a high probability that the same high frequency components in the blocks above and below are also zero. Therefore, the reconstruction scheme works accurately without the content of the image being known. Based on this heuristic, the weights of the reconstruction function

$$C_z = C_z + W_t \times C_t + W_b \times C_b + W_l \times C_l + W_r \times C_r$$

can be assigned based on which portion of the DCT block is being reconstructed:

- For the high frequency components in the horizontal scan (region A in Figure 12): $W_t = W_b = \frac{1}{2}$ and $W_l = W_r = 0$.
- For the high frequency components in the vertical scan (region B in Figure 12): $W_t = W_b = 0$ and $W_l = W_r = \frac{1}{2}$.
- For the rest of the coefficients: $W_t = W_b = W_l = W_r = \frac{1}{4}$.

Again, the advantage of this technique is its simplicity. The examples in Section 6.3 show its effectiveness.

Also note that, after the packetization scheme, not only does a block not lose all coefficients, but its neighboring blocks do not lose the same frequency components. This minimizes 1) the number of coefficients that need to be reconstructed in each block, and 2) the chance that consecutive blocks lose the same frequency components. Without this packetization scheme, reconstruction may interpolate using blocks many (e.g., 16 or 24) pixels away, blocks that may have no spatial correlation whatsoever.

6 Evaluation

We first present our study on memory use, which employed a *Quantum Viking* disk, a unit designed for desktop computers. We then evalueate the effectiveness of our image packetization and reconstruction scheme.

Parameter Name	*Value*
Disk Capacity	*10 GBytes*
Number of cylinders, CYL	*9,100*
Min. Transfer Rate TR	*98 Mbps*
Ultra2 SCSI (LVD) Transfer Rate	*640 Mbps*
Full Rotational Latency Time	*8.33 milliseconds*
Min. Seek Time	*0.9 milliseconds*
Max. Seek Time	*22.5 milliseconds*
$\alpha 1$	*0.82 milliseconds*
$\beta 1$	*0.08 milliseconds*
$\alpha 2$	*1.16 milliseconds*
$\beta 2$	*0.00234 milliseconds*

Figure 13: Quantum Viking II 9.1 WLS Disk Parameters

6.1 Memory Use

Given the *Quantum Viking* disk parameters, the only parameter that may be varied to tune IO performance is the page size p_{size}. Thus, we first experiment with how the page size affects memory use. We are also interested in finding out how much memory is needed to support a 19.2 Mbps DTV stream and how many 19.2 Mbps DTV streams the disk can support.

Figure 13 lists the parameters for the Viking disk. In addition, we assume that the peak data consumption and input rates are both 19.2 Mbps (a standard DTV bitrate). For computing the seek overhead we follow closely the model developed in [12, 16], which has been proven to be asymptotically close to the real disks. (We also wrote a test program to verify the disk performance. The actual disk performance is slightly better than specified.) The seek overhead function is the following concave function:

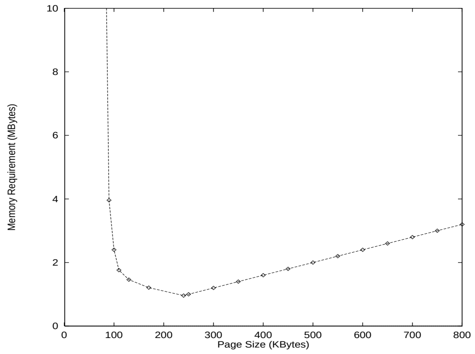
$$\gamma(d) = \alpha 1 + (\beta 1 \times \sqrt{d}) + 11.2 \text{ if } d < 900$$

$$\gamma(d) = \alpha 2 + (\beta 2 \times d) + 11.2 \text{ if } d \geq 900.$$

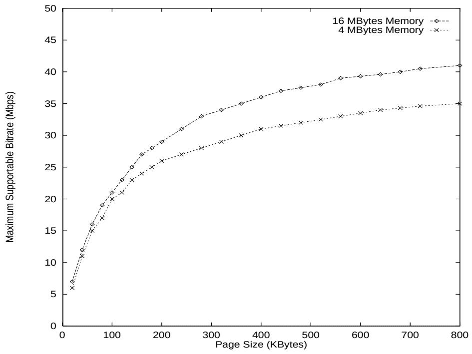
6.2 Page Size

Figure 14 shows the effect of page size on required memory and on peak input rate. The horizontal axis represents the memory page size up to 800 KBytes. (Page sizes beyond 800 KBytes convey no additional information.) Figure 14(a) shows the minimum memory requirement for a peak input rate of 19.2 Mbps for different page sizes. The minimum memory required to support a 19.2 Mbps DTV stream is about one MBytes when the page size is set between 240 and 250 KBytes. This surprisingly small memory requirement shows that our client-based approach is indeed a cost-effective architecture: instead of requiring GBytes of memory to buffer data for time-shift operations, using a disk costs significantly less!

Similarly, Figure 14(b) shows the maximum data input rates that our client-based architecture can support given 4 and 16 MBytes of main memory. (Memory sizes beyond 16 do not help noticeably.) The peak bitrate that 4 MBytes of memory can support is 35 Mbps and the



(a) M_{avail} vs. p_{size}



(b) Max. Bitrate vs. p_{size}

Figure 14: The Effect of Page Size

peak bitrate that 16 MBytes of memory can support is 41 Mbps (about two DTV streams) when the page size is 800 KBytes.

We draw four observations from these results.

- Not all page sizes yield feasible solutions. Figure 14(a) shows that when $p_{size} < 80$ KBytes, no feasible control parameters exist to support the peak data consumption and input rates of 19.2 Mbps. Figure 14(b) confirms that the supportable peak rate is 19.2 Mbps when $p_{size} = 80$ KBytes.
- Figure 14(a) shows that the memory requirement exhibits a sharp knee between $p_{size} = 100$ and 240 KBytes. The minimum memory requirement drops drastically as p_{size} approaches 100 KBytes. A larger page size decreases the number of inter-block seeks and consequently leads to more efficient IOs and memory savings. However, as soon as the data that arrives during one write IO is able to fit into one single page, increasing page size further only wastes memory. This explains why the memory requirement goes up when $p_{size} > 240$ KBytes. Thus, we should avoid selecting a larger than necessary page size.
- The maximum data input rate the resource manager can handle goes up with the page size as shown in Figure 14(b). This improvement results from the reduction in the inter-block seek overhead.
- Increasing memory does not significantly increase the peak bitrate that the system can support. The gap between the peak bitrates that can be supported by 4 and 16 MBytes of memory is not significant as shown in Figure 14(b).

We have implemented the resource manager to support interactive VCR functions. The prototype is implemented on a Pentium II 450 MHz Window95 workstation with a Quantum Viking II disk. The prototype has four threads: one controls user interface, one receives incoming bits, one decodes bits and displays frames, and one performs IOs. The CPU scheduling overhead and thread context switches do prolong the worst-case time to perform IO, but not by much (by about 10 ms). Using two MBytes of memory space, the prototype supports a 19.2 Mbps stream with a 400 display screen without a glitch. Although the memory requirement is doubled compared to that in the above theoretical study, it is nevertheless very low. We are therefore convinced that our proposed architecture does work effectively and economically.

6.3 Image Reconstruction

In our image reconstruction experiment, we first compare a damaged image that does not distribute the loss of the frequency components to other blocks (Figure 15) with one that uses our packetization scheme (Figure 16). Both images were generated under a loss rate of 20%. The major difference is that in the blocks where DC is lost, one can still see high frequency components in Figure 16. Although both images are damaged badly, Figure 16 looks slightly better. (The submitted images are colored.)

Next, we compared the reconstructed images from the first set of damaged images using our reconstruction technique, which takes spatial characteristics into consideration. Figure 17 is the image reconstructed from Figure 15, and Figure 18 is that reconstructed from Figure 16. It is clear that Figure 18 has better visual quality. The distant trees and lamps in Figure 18 are almost lossless, as seen when Figure 18 is compared with the original image in Figure 19. The flower bed of the image in Figure 18 has much better continuity than that of Figure 17, which suffers from severe blocking effects. The distribution of the loss by the proposed packetization scheme significantly reduces the blocking effects for which DCT has often been criticized.

6.3.1 Limitations

This technique is effective as long as there is only one burst of errors and the duration of the burst is smaller than half the image size. If the duration of the error burst is larger than half the image size, the lost coefficients may not be recovered from the neighboring blocks, in which case the image quality degrades severely. In addition, when there are multiple bursts of errors, the chance that they cluster on the image is increased, and subsequently the reconstruction technique may perform less well. However, we argue that such extreme conditions are rare, and other schemes may also be less effective under these conditions.

Figures 20, 21, and 22 show the reconstructed images at the loss rates of 30%, 40%, and 50% respectively.

6.3.2 A Note on SNR

The SNR has been used to measure the image quality, but has been shown not necessarily to reflect the quality of an image [8]. In the proposed packetization scheme, since the errors that are supposed to concentrate in one DCT block are distributed to 64 DCT blocks, the noise affects 64 blocks, rather than just one single block, after the inverse DCT is performed. Thus, the SNR is always slightly lower in the proposed packetization scheme (by less than one *db* in PSNR). But human eyes are less sensitive to errors that are diluted and spread out than to errors concentrated on a few spots. Therefore, the slightly lower SNR should not be a concern in this case.

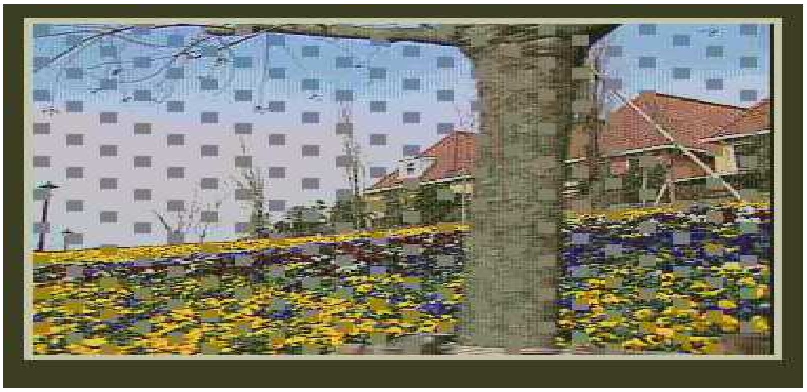


Figure 15: 20% Damaged Image Without Coefficients Spread Out

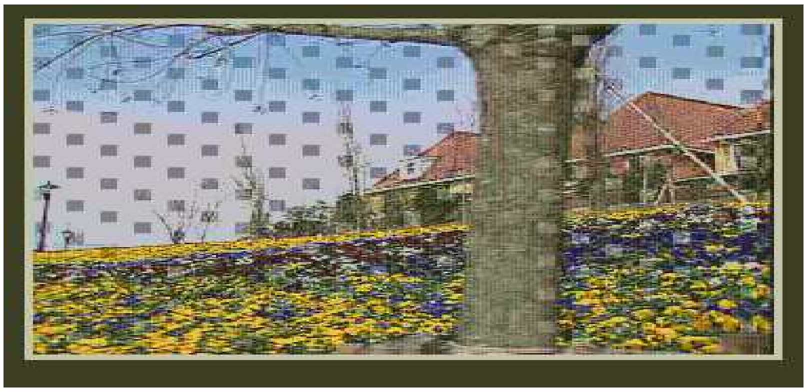


Figure 16: 20% Damaged Image With Coefficients Spread Out

7 Conclusion

We have presented a client-based architecture that supports interactive DTV features. Isolating the client from server and network variability and delay makes supporting "real-time" interactive DTV features possible. Moreover, our quantitative analysis and implementation demonstrated that with a large disk the memory use at the client side is minimum and hence the client-based architecture is cost effective.

In addition, we proposed techniques to maximize QoS at the client side. For video data, we proposed an error concealment and image reconstruction scheme that repairs damaged key frames. For the client as a whole, we proposed a scheduling policy that takes the characteristics of the data objects, the resource constraints, and the viewer's preference into consideration to maximize the total QoS.

In summary, a dynamic, effective and economical client-based architecture complements a good server design. Together the client and server can thus provide a complete end-to-end solution for the delivery of media data.

Figure 17: Image Reconstructed from Figure 15

Figure 18: Image Reconstructed from Figure 16

Figure 19: Original Undamaged Image

Figure 20: Reconstructed Image from 30% Loss

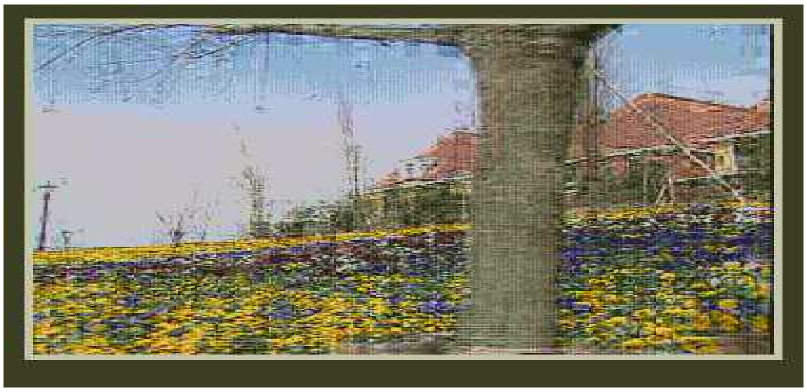


Figure 21: Reconstructed Image from 40% Loss

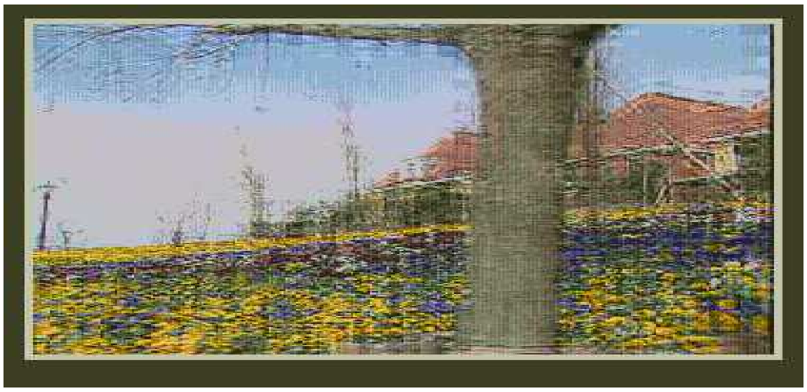


Figure 22: Reconstructed Image from 50% Loss

References

[1] H.261 specification. *ITU*, August 1996.

[2] E. Chang. An image coding and reconstruction scheme for mobile computing. *5th IDMS (Springer-Verlag LNCS#1483)*, pages 137–48, 1998.

[3] E. Chang and H. Garcia-Molina. Bubbleup - low latency fast-scan for media servers. *Proceedings of the 5th ACM Multimedia Conference*, pages 87–98, November 1997.

[4] E. Chang and H. Garcia-Molina. Effective memory use in a media server. *Proceedings of the 23rd VLDB Conference*, pages 496–505, August 1997.

[5] E. Chang and H. Garcia-Molina. Medic: Memory and disk cache for media servers. *Stanford Technical Report SIDL-WP-1997-0076 URL: http://www-diglib.stanford.edu (To appear in ICMCS'99)*, November 1997.

[6] E. Chang and H. Garcia-Molina. Reducing initial latency in media servers. *IEEE Multimedia*, 4(3):50–61, July-September 1997.

[7] I. Dalgic and F. Tobagi. Characterization of quality and traffic for various video encoding schemes and various encoder control schemes. *Stanford Technical Report CSL-TR-96-701*, August 1996.

[8] I. Dalgic and F. Tobagi. Performance evaluation of ethernets and atm networks carrying video traffic. *Stanford Technical Report CSL-TR-96-702*, August 1996.

[9] S. Ghandeharizadeh. Continuous retrieval of multimedia data using parallelism. *IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING*, 5(4):658–69, 1993.

[10] S. Hemami and R. Gray. Reconstruction-optimized lapped orthogonal transforms. *IEEE International Conf. on Acoustics Speech and Signal Processing*, 3:1542–45, 1996.

[11] S. Hemami and T. Meng. Transform coded image reconstruction exploiting interblock correlation. *IEEE Transaction on Imagine Processing*, 4(7):1023–27, July 1995.

[12] D. Kotz, S. B. Toh, and S. Radhakrishnan. A detailed simulation model of the hp 97560 disk drive. *Dartmouth College Technical Report PCS-TR94-220*, 1994.

[13] S. Lin and D. Costello. *Error Control Coding: Fundamentals and Applications*. Prentice-Hall, 1983.

[14] T. Meng. Portable video-on-demand in wireless communication. *Proceedings of the IEEE*, 83(4), April 1997.

[15] B. Ozden, R. Rastogi, and A. Silberschaz. *The Storage and Retrieval of Continuous Media Data, a Chapter in Multimedia Database Systems*. Springer, 1996.

[16] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *Computer*, 2:17–28, 1994.

[17] K. Sayood. *Introduction to Data Compression*. Morgan Kaufmann, 1996.

[18] S. B. Wicker and M. J. Bartz. The design and implementation of type-i and type-ii hybrid-arq protocols based on first-order reed-muller codes. *IEEE Transaction on Communications*, 42(2/3/4), 1994.