Pervasive and Mobile Computing 6 (2010) 607–622

Contents lists available at ScienceDirect

Pervasive and Mobile Computing

journal homepage: www.elsevier.com/locate/pmc

ChunkStream: Interactive streaming of structured data

Justin Mazzola Paluska*, Hubert Pham, Steve Ward

MIT Computer Science and Artificial Intelligence Laboratory Cambridge, MA, USA

ARTICLE INFO

Article history:
Received 10 April 2010
Received in revised form 19 July 2010
Accepted 27 July 2010
Available online 6 August 2010

Keywords:
Chunks
Video editing
Cloud computing
Video streaming

ABSTRACT

We present ChunkStream, a system for efficient streaming and interactive editing of online video. Rather than using a specialized protocol and stream format, ChunkStream makes use of a generic mechanism employing *chunks*. Chunks are fixed-size arrays that contain a mixture of scalar data and references to other chunks. Chunks allow programmers to expose large, but fine-grained, data structures over the network.

ChunkStream represents video clips using simple data types like linked lists and search trees, allowing a client to retrieve and work with only the portions of the clips that it needs. ChunkStream supports resource-adaptive playback and "live" streaming of real-time video as well as fast, frame-accurate seeking; bandwidth-efficient high-speed playback; and compilation of editing decisions from a set of clips. Benchmarks indicate that ChunkStream uses less bandwidth than HTTP Live Streaming while providing better support for editing primitives.

1. Introduction

Users increasingly carry small, Internet-enabled computers with them at all times. Some of these small computers are highly capable smartphones like Apple's iPhone or Google's Android, while others are lightweight netbooks.

These computers are "small" in the sense that they have a small form factor as well as smaller than normal processing and storage abilities. Nonetheless, users expect the same functionality from them as their full-sized brethren. For example, netbook users run full-fledged operating systems and applications on their machines while smartphone users may use full-featured applications to edit photos and spreadsheets directly on their phone.

Such small machines may be computationally overwhelmed by "big" tasks like editing video or creating complex documents. Luckily, many of those big tasks are centered around highly structured data types, giving us an opportunity to present large data structures in smaller units that impoverished clients may more easily consume and manipulate. At the same time, a common network protocol for expressing structure may allow us to share work between small clients and cloud-based clusters, making use of always-on network connections to compensate for poor computational abilities.

In this paper, we explore one way of exposing and sharing structured data across the Internet, in the context of cloud-based video editing. We choose to explore video editing because video is highly structured – video files are organized into streams, which are further organized into groups of pictures composed of frames – yet there is no existing protocol for efficient editing of remote video. Additionally, many video editing operations (such as special effects generation) are computationally intensive, and as such may benefit from a system where clients can offload heavy operations to a cluster of servers. Finally, video is already an important data type for mobile computing because most small computers include special support for high-quality video decoding, and now commonly even video capture.

* Corresponding author.
E-mail address: jmp@mit.edu (J. Mazzola Paluska).

1574-1192/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.pmcj.2010.07.006

1.1. Pervasive video editing

Just as users capture, edit, mashup, and upload photos without touching a desktop computer, as video capabilities become more prevalent, we expect that users will want to edit and mashup videos directly from their mobile devices. However, currently, video editing is a single-device affair. Users transfer all of their clips to their "editing" computer, make all edits locally, and then render and upload their finished work to the web. In a modern pervasive computing environment, users should be able edit videos "in the cloud" with whatever device they may have at the time, much as they can stream videos from anywhere to anywhere at anytime.

Video editing is a much more interactive process than simple video streaming: whereas users of video streaming engage in few stream operations [1], the process of video editing requires extensive searching, seeking, and replaying. For example, a video editor making cutting decisions may quickly scan through most of a clip, but then stop and step through a particular portion of a clip frame-by-frame to find the best cut point. After selecting a set of cut points, the editor may immediately replay the new composite video to ensure that his chosen transitions make sense. In another work flow, an editor may "log" a clip by playing it back at a higher than normal speed and tagging specific parts of the video as useful or interesting. Each of these operations must be fast or the editor will become frustrated with the editing process.

1.2. Internet-enabled data structures

Existing streaming solutions generally assume that clients view a video stream at normal playback speeds from the beginning [2] and are ill-suited to the interactivity that video editing requires. A challenge to enabling pervasive video editing is allowing small clients to manipulate potentially enormous video clips.

Two general principles guide our approach. First, we expose the internal structures of uploaded video clips to the client so that each client can make decisions about what parts to access and modify. Second, we offer "smaller", less resource-intensive proxy streams that can be manipulated in concert with full-sized streams.

Most streaming systems already follow these principles using ad hoc protocols and specialized data formats. For example, Apple's HTTP Live Streaming [3] exposes videos to clients as a "playlist" of short ($\approx$10 s) video segments. Each segment may have individually addressable "variants" that allow clients to choose between an assortment of streams with different resource requirements.

While it is possible to design a specialized video editing protocol that adheres to these principles, we believe that the fine-grained interactivity required by video editing and the device-specific constraints imposed by each small client may be better served by a more generic framework that allows the client and server to decide, at run-time, how to export and access video data. To this end, rather than fixing the protocol operations and data formats, we explore an approach that uses generic protocols and data formats as a foundation for building video-specific network-accessible data structures.

1.3. Contributions

This paper presents ChunkStream, a system that allows frame-accurate video streaming and editing that is adaptive to varying bandwidth and device constraints. We propose the use of a generic primitive – individually addressable "chunks"– that can be composed into larger, but still fine-grained, data structures. Using chunks allows ChunkStream to reuse generic protocols and data structures to solve video-specific problems.

ChunkStream exposes video clips as a linked list of frame chunks. Each video clip may include multiple streams of semantically equivalent video at different fidelity levels, allowing resource-constrained clients to play and edit low-fidelity streams while more powerful cloud-based servers manipulate high-fidelity streams. To overcome network latencies inherent in cloud-based architectures, the linked list is embedded in a search tree that enables a client to quickly find individual frames for editing cut points without needing to download the entire video clip over the network.

Our goal in this paper is to explore how using generic structuring mechanisms may lead to streamable, flexible, and useful data structures. While we focus on video, we believe that using chunks to create structured data streams is generalizable to any structured data type.

2. Architecture

ChunkStream is built on top of a single data type—the chunk. Using chunks as a foundation, we construct composite data structures representing video streams and editing decisions.

2.1. Chunks

A chunk is a typed, ordered, fixed-size array of fixed-size slots. Each slot is typed and may be empty, contain scalar data, or hold a reference ("link") to another chunk. Every chunk is also annotated with a type. Chunks are stored on a central server and may be requested by clients over the network.

Chunk links are explicit and as such can be used to create large data structures out of networks of chunks. Each chunk is identified by an identifier determined by the creator of the chunk. The chunk identifier is guaranteed to be unique within

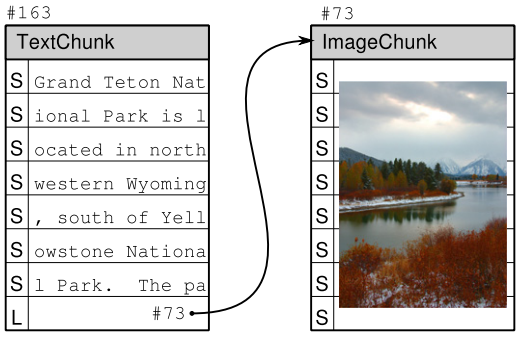

Fig. 1. Two chunks with a link between them. The gray bar indicates the start of the chunk and contains the type hint. Each slot is typed with a type field (S for scalar and L for links).

the namespace of the chunk server allowing clients to compare the identity of chunks by comparing identifiers. Chunk links are simply slots typed as references and filled with the identifier of the referent chunk.

The chunk type is a string that serves a specification of the semantics of the slots of the chunk. Applications and libraries that need to fully interpret a chunk must understand that chunk's type. However, this is not true of all code that accesses chunks: code that concerns itself with the chunk graph, such as "system-level" code like pre-fetching optimizations, may work simply by following the explicitly marked links without understanding the semantics of every slot.

Fig. 1 illustrates two chunks with $N = 8$ slots. The chunk on the left is a TextChunk, which has the specification that the first $N - 1$ slots are scalar Unicode text and that the last slot is a link to a related chunk. The TextChunk links to an ImageChunk, whose specification states that the first $N - 1$ slots are packed binary data of a photograph, and that the last slot is either scalar data or a link to another ImageChunk if the encoding of the photograph does not fit in a single chunk.

We chose chunks as our foundation data type for two reasons. First, since chunks contain a fixed number of fixed-size slots, there is a limit to the total amount of information that a single slot can hold as well as a limit to the number of chunks to which a single chunk can directly link. While fixed sizes may be seen as a limitation, size limits do have certain benefits. Small chunks force us to create fine-grained data structures, giving clients greater choices as to what data they want to access over the network. Moreover, since clients know the maximum sizes of the chunks they request, they can easily use the number of chunks in a data structure to account for total resource usage. Both features fit well with our requirements for video editing on small clients.

Our second reason for using chunks is that they export a flexible, reference-based interface that can be used to build, link together, and optimize larger data structures one block at a time. Chunks can be used to implement any pointer-based data structure, subject to the constraint that the largest single element fits in a single chunk. Chunks may also be used as unstructured byte arrays by simply filling slots with uninterpreted binary data. For example, as shown in Fig. 2, a larger version of the photograph in Fig. 1 may be split across a linked list of four ImageChunk chunks. Since the chunk interface is fixed between the client and the server, chunk-based data structures can be shared between the two without content format mismatches or undefined links.

Fig. 3 shows the chunk store API that the centralized server uses to create, modify, and persist chunks. Chunks must be inserted into the chunk store in order to create links between them. To create the data structure in Fig. 1, we first call new_chunk() to create the image chunk. After filling the chunk with image data, we call insert() to persist the chunk to the store; insert() returns an identifier (#73) to use to refer to the chunk. Next, we call new_chunk() a second time to create the text chunk. After filling the first seven slots with text data, we create the link in the eighth slot by filling the slot with the identifier of the target and marking the slot as a link. Finally, we call insert() to put the chunk into the store, which returns the chunk's identifier (#163).

A client that knows the identifier of a chunk uses get() to retrieve the chunk. Using Fig. 1 as an example, if a client has a local copy of chunk #163, then it may fetch chunk #73 by reading the contents of the last slot in chunk #163 and passing the slot's value to the get() call.

2.2. Representing video streams with chunks

Our video representation is guided by the observation that non-linear video editing consists of splicing and merging streams of audio and video together to make a new, composite stream that represents the edited video. If we were to represent each video stream as a linked list of still frames, editing would be the process of creating a brand new stream by modifying the "next" pointer of the last frame we want from a particular stream to point to the first frame of the subsequent stream. Adding a special effect, like a fade or wipe, would only require creating a new "special effect stream" and then linking it into our final video like any other stream.

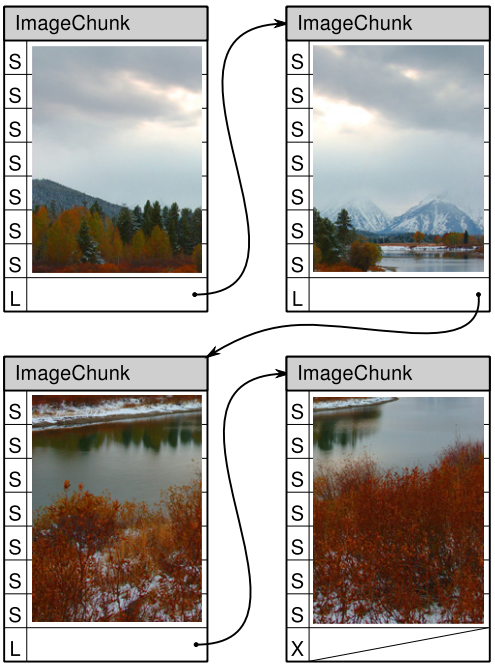

Fig. 2. Large objects must be split among many chunks. Here, a larger version of the image in Fig. 1 is partitioned into a linked list of four chunks. The last slot of the last chunk in the linked list is empty.

`new_chunk()` **returns** a new, uninitialized chunk
Create a new, unnamed chunk.

`insert(chunk)` **returns** chunk identifier
Insert `chunk` into the store and return the chunk's identifier.

`get(chunk_id)` **returns** a chunk
Return the chunk associated with the chunk identifier.

`modify(chunk_id, new_chunk)` **returns** void
Modify the chunk associated with the chunk identifier so that it has the contents of `new_chunk`.

Fig. 3. Chunk store API.

2.2.1. Video streams

We represent video streams with four types of chunks, as illustrated in Figs. 4 and 5. The underlying stream is represented as a doubly linked list of "backbone" chunks. The backbone is doubly linked to allow forward and backward movement within the stream. For each frame in the video clip, the backbone links to a "LaneMarker" chunk that represents the frame.

The LaneMarker chunk serves two purposes. First, it links to metadata about its frame, such as frame number or video time code. Second, the LaneMarker links to "lanes" of video. A lane is a video substream of a certain bandwidth and quality, similar to segment variants in HTTP Live Streaming. A typical LaneMarker might have three lanes: one high-quality HD lane for powerful devices and final output, a low-quality lane suitable for editing over low-bandwidth connections on small devices, and a thumbnail lane that acts as a stand-in for frames in static situations, like a timeline. ChunkStream requires that each lane be semantically equivalent (even if the decoded pictures are different) so that the client may choose the lane it deems appropriate based on its resources.

As shown in Fig. 5, each lane takes up two slots in the LaneMarker chunk. One of these lane slots contains a link to a StreamDescription chunk. The StreamDescription chunk contains a description of the lane, including the codec, stream profile information, picture size parameters, stream bit rate, and any additional flags that are needed to decode the lane. Lanes are not explicitly marked with identifiers like "HD" or "low-bandwidth". Instead, clients derive this information from the parameters of the stream: a resource-constrained device may bypass a 1920 × 1080 "High Profile" stream in favor of a 480 × 270 "Baseline" profile stream while a well-connected desktop computer may make the opposite choice. As an optimization, ChunkStream makes use of the fact that in most video clips, the parameters of the stream do not change over

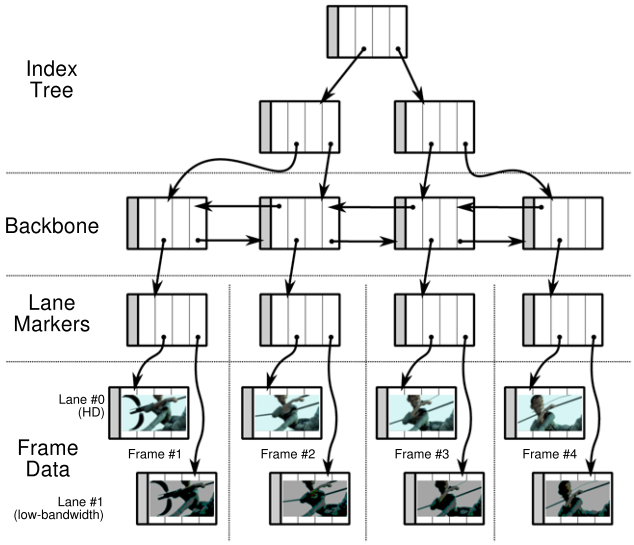

Fig. 4. Chunk-based video stream representation used by ChunkStream. LaneMarker chunks represent logical frames and point to "lanes" composed of FrameData chunks. Each lane represents the frame in different formats, e.g. using HD or low-bandwidth encodings. Links to StreamDescription chunks and unused slots at the end of chunks are not shown.

Type: LaneMarker		
Slot	Type	Description
0	Data	Frame number (presentation timestamp)
1	Link	Link to StreamDescription chunk describing Lane 1
2	Link	Link to FrameData chunk containing the picture data of Lane 1.
...		
$2i$	Link	Link to StreamDescription chunk describing Lane i+1
$2i$+1	Link	Link to FrameData chunk containing the picture data of Lane i+1

(a) LaneMarker chunks delineate different substreams.

Type: StreamDescription		
Slot	Type	Description
0	Data	Codec Name
1	Data	Picture width (as a string)
2	Data	Picture height (as a string)
3	Data	Codec-specific parameters
...	Data	Codec-specific parameters
N-1	Data	Stream Identifier (c.f. Section 2.6)

(b) StreamDescription chunks describe contain stream metadata.

Type: FrameData		
Slot	Type	Description
0	Link	Link to FrameContext chunk or empty if this is an I-frame (c.f. Section 2.4)
1	Data	Binary frame data
...		
N-2	Data	Binary frame data
N-1	Link	Link to the next FrameData chunk, or empty if no more chunks are needed.

(c) FrameData chunks contain binary data that makes up a frame.

Fig. 5. Chunk data formats used to represent video streams in ChunkStream. N is the fixed number of slots in the chunk.

the life of the stream, allowing ChunkStream to use a single StreamDescription chunk for each lane and point to that single chunk from each LaneMarker.

The other slot of the lane points to a FrameData chunk that contains the underlying encoded video data densely packed into slots. If the frame data does not fit within a single chunk, the FrameData chunk may link to other FrameData chunks.

Finally, in order to enable efficient, $O(\log n)$ random frame seeking (where n is the total number of frames), we add a search tree, consisting of IndexTree chunks, that maps playback frame numbers to backbone chunks.

2.2.2. Video playback

Listing 1 shows the algorithm clients use to play a video clip in ChunkStream. A client first searches through the IndexTree for the backbone chunk containing the first frame to play. From the backbone chunk, the client follows a link

Type: EDLClip		
Slot	Type	Description
0	Link	Link to the previous EDLClip, or empty if this is the first clip.
1	Link	Link to the next EDLClip, or empty if this is the last clip.
2	Data	Number of frames to play
3	Link	Link to the root of the IndexTree of the clip
4	Data	Start frame number

Fig. 6. EDLClip chunk format.

to the LaneMarker for the first frame, downloads any necessary StreamDescription chunks, and examines their contents to determine which lane to play. Finally, the client de-references the link for its chosen lane to fetch the actual frame data and decodes the frame. To play the next frame, the client steps to the next element in the backbone and repeats the process of finding frame data from the backbone chunk.

```
1  def play(tree_root, frame_num):
2
3      # Search through the IndexTree:
4      backbone = indextree_find_frame(tree_root, frame_num)
5
6      while playing:
7          # Dereference links from the backbone down to frame data:
8          lm_chunk = get_lane_marker(backbone, frame_num)
9          lane_num = choose_lane(lm_chunk)
10         frame_data = read_lane(lm_chunk, lane_num)
11         # Decode the data we have
12         deocde_frame(frame_data)
13
14         # Advance to next frame in backbone:
15         (backbone, frame_num) = get_next(backbone, frame_num)
```

Listing 1: Client-side playback algorithm for ChunkStream.

An advantage of exposing chunks directly to the client is that new behaviors can be implemented without changing the chunk format or the client/server communication protocol. For example, we may extend the basic playback algorithm on the client with additional, editing-friendly commands by simply altering the order and number of frames we read from the backbone. In particular, a client can support reverse playback by stepping backwards through the backbone instead of forwards. Other operations, like high-speed playback in either direction may be implemented by skipping over frames in the backbone based on how fast playback should proceed.

It is also possible to implement new server-side behaviors, such as live streaming of real-time video. Doing so simply requires dynamically adding new frames to the backbone and lazily updating the IndexTree so that new clients can quickly seek to the end of the stream.

2.2.3. Editing

Clients compile their desired cuts and splices into an Edit Decision List (EDL). The EDL is a doubly linked list of EDLClip chunks. Each EDLClip chunk references the IndexTree of the clip, the frame at which to start playing, and the length the clip should play. Fig. 6 details the EDLClip format. The order of the EDLClips in the linked list indicates the order in which to play the video clips. EDLs are lightweight since each EDLClip chunk references frames contained within an existing ChunkStream clip rather than copying frames. They are also easy to modify since new edit decisions can be added or existing decisions can be removed by changing the linked list. Using EDLs also allows ChunkStream to make video clips themselves immutable, aiding optimization and caching (cf. Section 2.5).

Fig. 7 illustrates an EDL and a series of shots between two characters engaged in a dialog. The EDL references video from clip 1 for 48 frames and then clip 2 for 96 frames. To play through the EDL, the client loads the clip referenced by the first EDLClip in the EDL, seeks to the correct position in the clip and then plays it for the specified number of frames before moving to the next EDLClip chunk in the EDL and repeating the process.

2.3. Editing example

A user that edits with ChunkStream first uploads his video clips to the ChunkStream server, which creates the relevant IndexTree and backbone structures. If possible, the server transcodes the clip to create less resource-intensive lanes for small clients. For each uploaded clip, the server returns a reference to the root of that clip's IndexTree. In addition to the clips he uploaded, the user may also use other clips that are on the ChunkStream server. To do so, the user's client only needs a reference to the root of the clip's IndexTree, potentially provided by a search feature on the server.

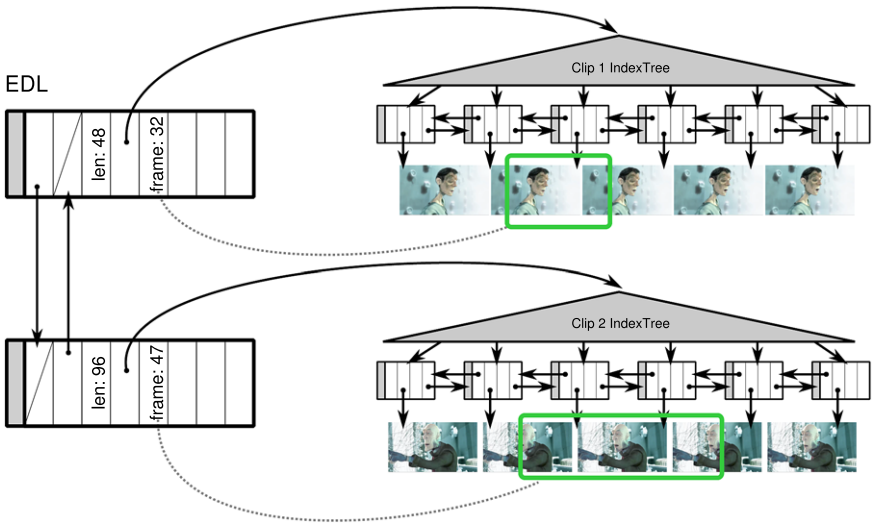

Fig. 7. An Edit Decision List (EDL) using portions of two clips.

After uploading the clips, the user scans through the clips he wants to use. In order to provide the user with an overview of each clip, the user's client runs through the backbone and decodes every 60th frame of a low-quality lane to create a filmstrip of thumbnails. After reviewing the thumbnail filmstrips, the user starts playing portions of each clip. As he marks start and stop points on the clips, his client compiles an EDL that points to the different parts of each clip that the user would like to use. Occasionally, the user changes the start and stop points or re-orders the clips; in response, his client just modifies the EDL.

The EDL is saved on the server, allowing the user to switch clients and still access his work. If the user switches to a resource-constrained client like a smartphone, the client will make use of lower-quality lanes knowing fully well that any edits he makes will also be reflected in the higher-quality lanes on the server.

After running through a few iterations, the user is happy with the rough cut of his video and starts adding special effects like fades. Rather than modifying the existing clips to add special effects, the user's client asks the ChunkStream server to create a new clip containing the special effect. The server creates the new clip, including all relevant IndexTrees and alternative lanes, and passes a reference to the new clip's IndexTree root. The client then integrates the special effect clip by modifying the EDL.

Finally, when the user is ready to make the final cut of his video, the client asks the server to compile the EDL to a new ChunkStream clip. In order to do so, the server reads through the EDL and copies the relevant frames (transcoding them as necessary) to a new IndexTree and backbone. The new video can then be viewed by and shared with other users on the ChunkStream server.

While editing, ChunkStream actively encourages reuse and referencing of already uploaded data. It is only during the creation of new content, such as new special effects or the final "playout" step that new IndexTree, backbone, and FrameData chunks are created. During all other steps, the client just refers to the already existing chunks, and uses ChunkStream's fast seeking features to make it appear to the user that his edits have resulted in a brand new clip on the server.

2.4. Codec considerations

In traditional video files, raw data is organized according to a container format such as MPEG-TS, MPEG-PS, MP4, or AVI. ChunkStream's IndexTrees, backbones, and LaneMarkers serve to organize raw encoded streams and, as such, may be considered as a specialized container format.

Container formats offer many advantages over raw codec streams. Most formats allow a single file to contain multiple streams, such as a video stream and one or more audio streams, as well as provide different features depending on the environment. Some formats, like MPEG-TS, include synchronization markers that require more bandwidth, but tolerate packet loss or corruption, while others are optimized for more reliable, random-access media. However, at its core, the job of the container format is to present encoded data in the order in which the decoder needs it.

This is especially important for video codecs that use inter-frame compression techniques because properly decoding a particular frame may depend on properly decoding other frames in the stream. For example, the H.264 codec [4] includes intra-frames (I-frames) that can be decoded independently of any other frame; predictive frames (P-frames) that depend on the previously displayed I- or P-frames; and bi-predictive frames (B-frames) that depend on not only the previously displayed I- or P-frames, but also the *next* I- or P-frames to be displayed. P-frames are useful when only a small part of the picture changes between frames, e.g., a clip of a talking head. B-frames are useful when picture data needed for the current frame is available from later frames in the clip, e.g., in a clip where a camera pans across a landscape or background scenery flows past a moving vehicle.

Type: FrameContext		
Slot	Type	Description
0	Link	Link to first FrameData chunk of the first dependency frame
	...	
i	Link	Link to first FrameData chunk of the i+1 dependency frame

Fig. 8. FrameContext chunk format.

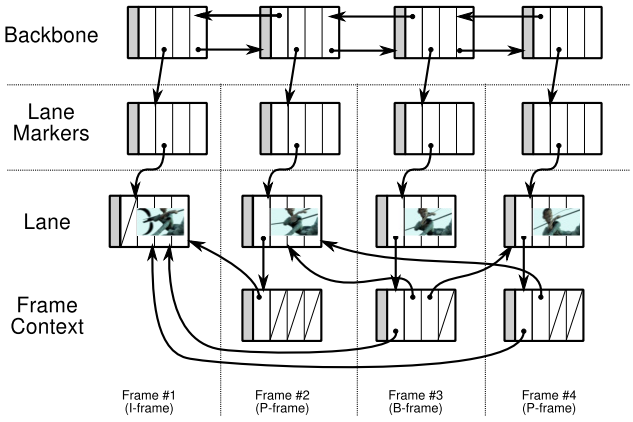

Fig. 9. FrameContext chunks allow inter-frame compression by exposing the dependencies of a particular frame. The first slot in the FrameContext chunk always points to the I-frame that serves as the foundation of the dependent group of pictures and subsequent slots point to frames in the dependency chain between the I-frame and the current frame in the required decode order.

An H.264 stream that is encoded using P- and B-frames is substantially smaller at the same quality level than a stream encoded using only I-frames. Unfortunately for video editing, using inter-frame compression complicates seeking, high-speed playback, and inter-clip splicing because frames are no longer independent and instead must be considered in the context of other frames. For example, a client that seeks to a P- or B-frame and displays only that frame without fetching other frames in its context will either not be able to decode the frame or decode a distorted image. Moreover, in codecs like H.264 with bi-predictive frames, the order in which frames are decoded is decoupled from the order in which frames are displayed, and the container format ensures that the decoder gets all the data it needs (including out-of-order frames) in order to decode the current frame. In other words, a container format contains the raw encoded streams in "decode order" rather than display or "presentation order".

Frames in a ChunkStream backbone are always in presentation order. ChunkStream uses presentation order because each lane in a stream may be encoded with different parameters or even different codecs, potentially forcing each lane to have a different decode order. To help clients determine the frame decode order, ChunkStream augments the FrameData chunks that make up each lane with FrameContext chunks that specify the dependencies of each frame.

Fig. 8 illustrates the FrameContext chunk format. Each FrameContext chunk contains a link to each FrameData chunk that the frame depends on, one link per chunk slot, in the order that the frames must be decoded. FrameContext chunks contain the full context for each frame, so that the frame can be decoded without consulting any other frame's FrameContext chunks. As a consequence, the first slot of a FrameContext chunk always points to the previous I-frame in presentation order.

For example, in Fig. 9, we show a portion of an H.264-encoded video clip using inter-frame compression. Since I-frames can be decoded independently, frame 1 has no FrameContext chunk. Frame 2 is a P-frame, and as such, depends on frame 1. Frame 2, therefore, has a FrameContext chunk that specifies that frame 1 must be decoded before frame 2 can be decoded. Similarly, frame 4 has a FrameContext chunk that specifies that frame 2, and as a consequence, frame 1 must be decoded before it can be decoded.[1] Finally, frame 3 is a B-frame and as such depends on both frame 2 and frame 4, and by dependency, frame 1, so its FrameContext chunk lists frames 1, 2, and 4.

2.5. Overheads and optimizations

The downside of breaking up a video into small chunks is that each chunk must be requested individually from the server, leading to an explosion of requests over the network. There are three ways we can mitigate this request overhead: (1) densely packed infrastructure chunks, (2) caches, and (3) server-side path de-referencing.

[1] In order to prevent circular references, H.264 does not allow P-frames to depend on B-frames, but only on previous P- or I-frames. As such, frame 4 cannot depend on frame 3.

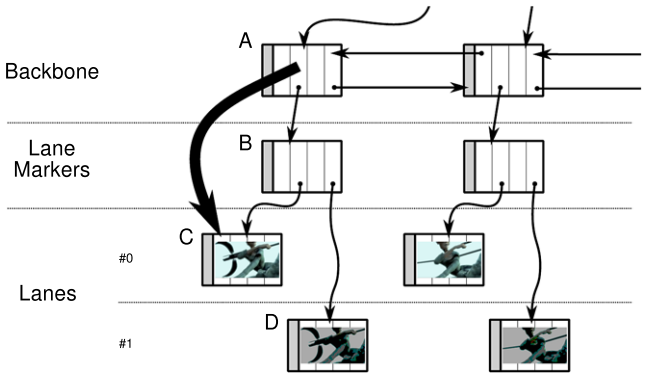

Fig. 10. Clients using server-side path de-referencing may skip over LaneMarkers if they know which lane they want to use.

2.5.1. Densely packed infrastructure chunks

IndexTree and backbone chunks are needed in order to create large, accessible data structures out of small chunks. Since they carry no video information, they are pure overheads. One way to reduce the number of IndexTree and backbone chunks is to use all available slots to create *n*-ary search trees and linked lists with fat nodes.

2.5.2. Caches

Densely packed infrastructure chunks are particularly useful when combined with client-side caching because the client will need to repeatedly access the same infrastructure chunks as it plays through a series of frames. Caches are also helpful when fetching the context chunks for a frame, since many frames may have the same, or similar, contexts.

By using a cache, the client may be able to skip many repeated network transfers and reduce the total load on the server and network connection. ChunkStream's cache coherency protocol is very simple: indefinitely cache IndexTree, backbone, and FrameData chunks from video clips, since they never change after being uploaded; cache LaneMarker chunks with a server-specified timeout since the server may modify the LaneMarker to add additional lanes as processing time allows; and never cache EDLClip chunks, since they are volatile.

2.5.3. Server-side path de-referencing

In lines 8 and 9 of our playback algorithm in Listing 1, clients fetch the lane marker and use it to determine which lane to play. If a client has determined that a particular lane of a stream meets its requirements, it will always follow the same general path from the backbone through the LaneMarkers to the underlying video frames.

Since the client does not need the intermediate LaneMarker, except to use it to follow links to FrameData chunks, we may lower request overheads by giving the chunk server a path to de-reference locally and send the client only the terminal chunk of the path. Fig. 10 shows a sample stream where the client has chosen to read the stream in the first lane. Rather than requesting chunk A, then B, then C, it may request that the server de-reference the slot 1 in chunk A, and then de-reference slot 1 in chunk B, and only return C, saving the transfer of chunks A and B. Server-side path de-referencing is beneficial because it has the potential to save both bandwidth and network round trips.

2.6. Multimedia and parallel streams

So far, we have concentrated only on video streams within the ChunkStream data structures. However, most "videos" are multimedia and contain more than just video streams: a typical video file also normally contains at least one audio stream and occasionally metadata streams like subtitles. These additional streams compliment the original video and should be played with the chosen video stream.

More generally, we may consider each of the substreams of a multimedia stream to be a part of a *parallel stream*. Parallel streams present two new challenges. First, different parallel streams from disparate sources may be mixed together, e.g., an editor may choose to mix the video from one camera with the audio from a boom microphone or music track. Second, parallel streams must be synchronized with each other. For example, audio and video streams must be synchronized to within 80 ms of each other in order for most people to believe that the streams are "lip-synced" [5].

We represent parallel streams by allowing LaneMarker chunks to contain lanes from multiple semantic streams. Recall that each lane of a LaneMarker chunk contains a link to a StreamDescription chunk with a stream identifier (Fig. 5(b)) as well as a link to the actual data in that lane. ChunkStream clients use the stream identifier slot of StreamDescription chunks to determine which lanes are alternate versions of the same stream (two or more lanes have identical stream identifiers) and which lanes are parallel streams (two lanes have different stream identifiers).

In order to allow editors to create parallel streams from disparate ChunkStream clips, we extend EDLClip chunks to reference additional lanes and stream identifiers as shown in Fig. 11. Clients that read an EDLClip with parallel streams must start playing all of the parallel streams at the same time to ensure that the streams are synchronized.

Type: EDLClip		
Slot	Type	Description
0	Link	Link to the previous EDLClip, or empty if this is the first clip.
1	Link	Link to the next EDLClip, or empty if this is the last clip.
2	Data	Number of frames to play
3	Link	Link to the root of the IndexTree of the clip for parallel stream 1.
4	Data	Start frame number for parallel stream 1.
5	Data	Stream identifier for parallel stream 1.
		...
$3i+3$	Link	Link to the root of the IndexTree of the clip for parallel stream $i+1$.
$3i+4$	Data	Start frame number for parallel stream $i+1$.
$3i+5$	Data	Stream identifier for parallel stream $i+1$.

Fig. 11. EDLClip chunk format extended to support multiple parallel streams. The changes compared to the EDLClip of Fig. 6 are highlighted in gray.

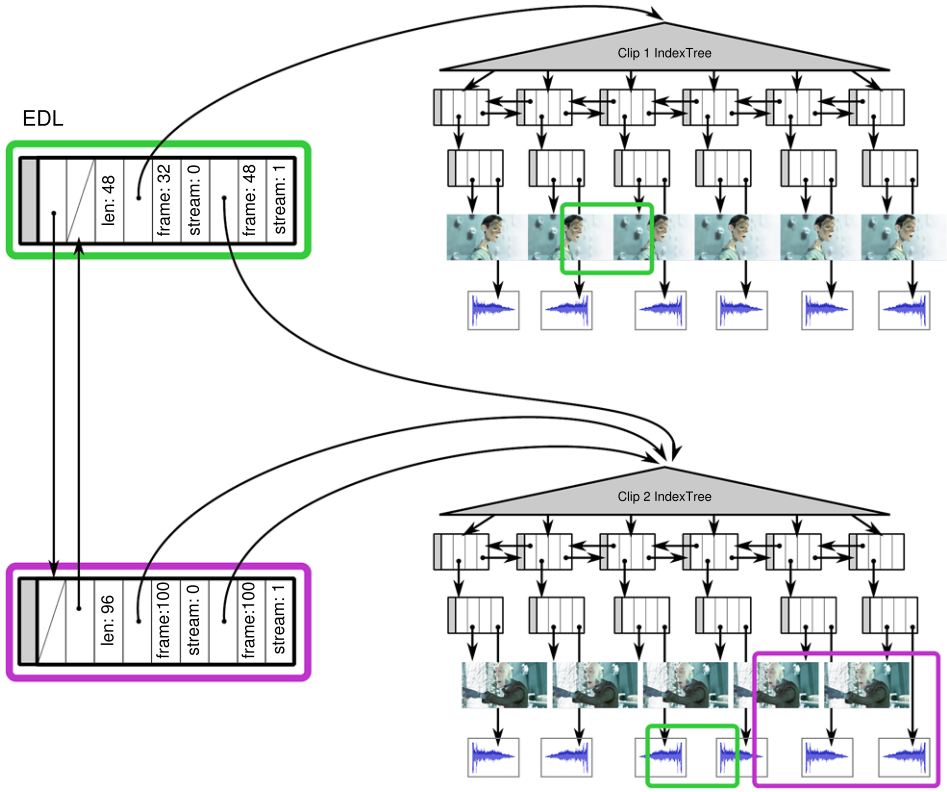

Fig. 12. Editing using parallel streams. Here, the first EDLClip combines video from the first clip with audio from the second, followed by one that incorporates both audio and video entirely from the second clip.

ChunkStream requires that each stream in an EDLClip chunk play for the same amount of time, so if the editor wants to change the audio stream while playing the same video stream, the editor's client must split the EDLClip into two EDLClip chunks, one with the first audio stream and the first portion of the video, and one with the second audio track and the latter part of the video.

Fig. 12 shows an example of mixing audio and video streams. In this example, we have two multimedia streams representing different camera angles in a conversation between two users. The first EDLClip chunk combines video from the first clip with audio from the second clip, both for 48 frames. The following EDLClip references 96 frames of video and audio from the same clip.

3. Implementation

In order to test our ChunkStream ideas, we built a prototype server and client library. Our implementation is divided into two main parts: a generic chunk-based foundation and a set of specialized libraries that implement ChunkStream-specific functionality. We implemented all of the libraries in Python 2.6 and tested our implementation on GNU/Linux and Mac OS X. All source code is available under a free license from http://o2s.csail.mit.edu/chunks.

3.1. Chunk implementation

The chunk implementation includes a chunk store library, a data structure library, a chunk server that exposes chunks over HTTP, and client libraries that understand the server's HTTP-based protocol.

3.1.1. Chunk store

The chunk store is responsible for persisting chunks on disk and supporting the chunk API of Fig. 3. In our simple implementation, chunks are stored as files, though the chunk API may admit many other implementations that are more efficient or have different consistency properties. The chunk store uses monotonically increasing integers to name chunks and ensures that each name is only used once, even if a particular chunk is garbage collected from the system.

3.1.2. Data structure library

Data structures are central to efficient use of the chunk model. To that end, our chunk implementation includes library functions to create common data types like densely packed linked lists and n-ary search trees, as well as functions to efficiently retrieve data from the same data structures. The IndexTree and backbone chunks of ChunkStream clips are just instances of structures from our library. The library also includes functions to pack opaque binary data into chunks.

3.1.3. Server and client

The server exposes chunks over HTTP and may run in any web server that supports CGI and Python. The HTTP interface allows clients to read the contents of chunks using standard HTTP GET requests (e.g., `GET /chunks/01234`), create chunks using HTTP POST requests, modify the contents of chunks using HTTP PUT requests.

Our server also supports the server-side path de-referencing optimization of Section 2.5.3. Clients request server-side de-referencing by adding a series of path segments to the end of their GET path. For example, a client may request chunk C from Fig. 10 using `GET /chunks/A/1/1`.

The CGI library is only required for stores that allow clients to change chunks, e.g., as needed by a web-based video editing applications, or by services that wish to make use of server-side path de-referencing optimization. Stores that simply expose a set of pre-computed chunks could be equally well served by a standard web server serving the static files saved by the chunk store.

The chunk client library implements the chunk store API of Fig. 3 by contacting a server over HTTP. The library also contains a client-side chunk cache.

3.2. ChunkStream

The ChunkStream-specific part of our codebase is split into two pieces: (1) utility code to create ChunkStream clips and EDLs and (2) libraries that aid playback and editing. The code to create ChunkStream clips uses FFmpeg [6] to break H.264 streams into its constituent frames, then makes use of our data structure library to create the relevant parts of the ChunkStream clip data structure. The playback and editing code implements a few Python classes that client applications may instantiate to playback ChunkStream clips or create and playback EDLs.

Our source distribution contains two example applications that use the ChunkStream libraries. The first, `simple_server.py` is a wrapper around our server library. `simple_server.py` takes, as input, a set of files to serve as ChunkStreams, decodes them using FFmpeg, and serves them over HTTP. The second application, `decode_client.py` reads ChunkStreams from a ChunkStream server, decodes the video, and write the individual frames as images to disk.

4. Evaluation

In order to show that our approach is useful, we must show that ChunkStream is competitive with other approaches in the resources that it uses. In particular, we hypothesize (1) that video streaming over ChunkStream is no worse than existing solutions and (2) that interactive editing operations under ChunkStream perform better than existing solutions.

To provide context for our benchmarks, we compare ChunkStream to two other approaches: downloading full files via HTTP and HTTP Live Streaming. For the ChunkStream tests, we use our server and client libraries. Our HTTP Live Streaming benchmarks use a custom Python library that parses HTTP Live Streaming `.m3u8` playlist files and queues relevant files for download. In all benchmarks, HTTP traffic passes through an instrumented version of Python 2.6's `urllib2` standard library module that allows us to capture the number of bytes transferred by each client (both upstream and downstream).

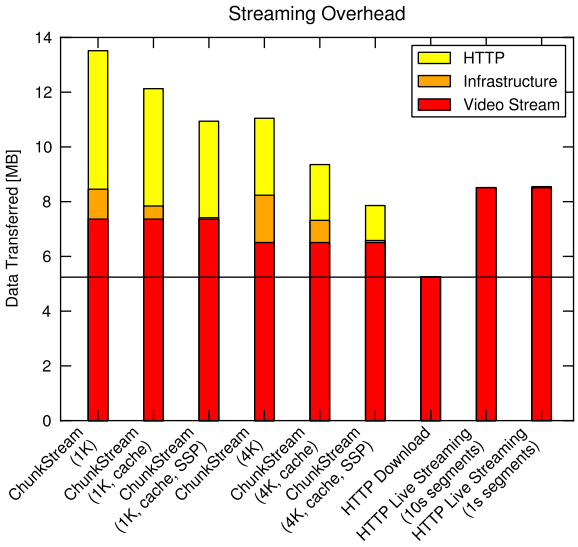

Fig. 13. Data transferred streaming a video using ChunkStream, standard downloading, and HTTP Live Streaming. The horizontal line is the size of the raw video stream.

For all of our tests, we used the first minute of the 720 × 405 24 frame/second H.264-encoded version of *Elephants Dream* [7] as the source video. We chose *Elephants Dream* for its permissive Creative Commons Attribution license and high-definition source videos. We removed the audio tracks from the video in order to concentrate solely on video streaming performance.

Scripts to replicate our benchmark tests are available with the ChunkStream source code.

4.1. Streaming performance

In order to evaluate how efficiently ChunkStream streams video, we measure the total amount of data transferred by the client (both upstream and downstream) as it streams our sample video. We do not make time-based streaming measurements because our benchmark assumes that the client has sufficient bandwidth to stream the video without dropping frames.

Fig. 13 shows the total amount of data transferred using ChunkStream, HTTP download, and HTTP Live Streaming. We classify the bytes transferred into three categories. The first category, "Video Stream", represents the video as encoded and streamed by the system. Next, "Infrastructure" represents bytes dedicated to infrastructure concerns, such as playlists in HTTP Live Streaming or IndexTrees and backbones in ChunkStream. Finally, "HTTP" represents bytes that are solely due to the HTTP protocol exclusive of HTTP bodies, such as request and response headers.

In Fig. 13, the first three ChunkStream bars show results using 1 kB sized chunks (containing 32 slots of 32 bytes each); the next three show measurements using 4 kB sized chunks (64 slots of 64 bytes each). In this particular case, 4 kB chunks have lower overhead than 1 kB chunks since the client must download fewer chunks and because the backbone chunks may contain more outgoing links per chunk.

Within each size category of chunks, we show measurements with our cache (bars marked "cache") and server-side path de-referencing (bars marked "SSP") optimizations turned on. Caching reduces both the Infrastructure and HTTP overheads by reducing the number of Infrastructure chunks that must be read in order to play the video. The cache is useful because backbone chunks are densely packed and contain many pointers to LaneMarkers and would otherwise be read over the network repeatedly. Server-side path de-referencing allows the client to avoid reading LaneMarkers. With both caching and server-side path de-referencing enabled, Infrastructure overheads weigh in at about 40 kB, negligible compared to the size of the video.

HTTP Download is the best case scenario since the downloaded file is simply a small MP4 wrapper around the raw H.264 stream. Unfortunately, a file in such a format typically cannot be viewed until it is completely downloaded. HTTP Live Streaming uses MPEG-TS as its container format. MPEG-TS is optimized for lossy channels like satellite or terrestrial broadcast and includes protections for lost or damaged packets. Unfortunately, such protections are unnecessary for TCP-based protocols like HTTP and lead to a 20%–25% overhead compared to the raw video. In the optimized case with caching and server-side path de-referencing enabled, ChunkStream carries 15% overhead compared to the raw video, which is better than HTTP Live Streaming and acceptable for consumer applications.

4.2. Editing operation performance

Next, we measure how interactive editing operations under ChunkStream compare to existing solutions. In this section, for the ChunkStream tests, we use 4 kB chunks with server-side path de-referencing and caching enabled.

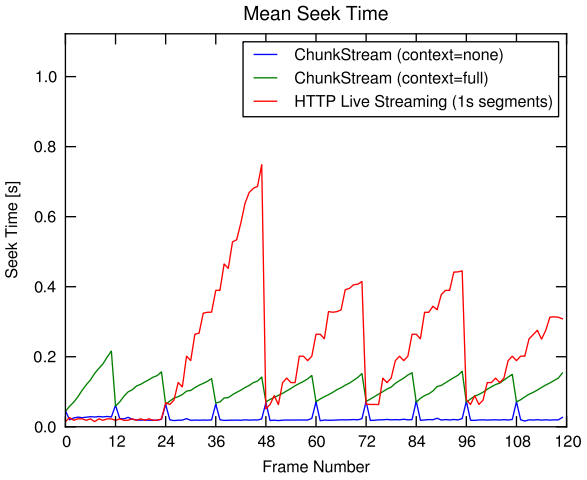

Fig. 14. Mean seek time to seek to a random position in the stream. The clients are on a network limited to wireless 3G-like speeds of 2 Mbit/s.

4.2.1. Frame-accurate seeking

Fig. 14 shows the amount of time it takes to seek to a specified frame in the video clip using a cold cache. To mirror the network environment small devices may encounter, we use trickle [8] to simulate the optimal 2Mbit/s bandwidth of wireless 3G networks.

We measure ChunkStream seek times under two different conditions: fetching only the sought-after frame without context (context=none) and fetching the entire context to accurately decode the sought-after frame (context=full). For HTTP Live Streaming, we test with 1 s segments, the minimum allowed by the HTTP Live Streaming IETF draft. We do not show HTTP Download as that requires downloading the entire file, which takes around 50 s at 3G network speeds.

Seek time for ChunkStream is, in general, lower than HTTP Live Streaming. The seek times for ChunkStream and HTTP Live Streaming both form periodic sawtooth curves. The period of the HTTP Live Streaming curve is equal to the segment size of the stream, in this case 1 s, or 24 frames. The curve is a sawtooth because frames in an HTTP Live Streaming MPEG-2 Transport Stream are of variable size so a client must scan through the segment to find the frame it needs, rather than requesting specific byte ranges from the server. The period of the ChunkStream curves is set by the frequency of I-frames in the underlying H.264 stream. *Elephants Dream* is encoded with an I-frame every 12 frames, which shows up as spikes in the context=none measurements. The sawtooth of the context=full measurements comes from having to fetch all frames between the I-frame and the sought-after frame.

Note that HTTP Live Streaming is much more efficient in the first second (first 24 frames) of the video. This is because the first second of *Elephants Dream* is composed of only black frames, making the first HTTP Live Segment only 5 kB in length and a very quick download. Subsequent segments are much larger, and as a consequence, it takes much more time to seek within the HTTP Live Stream. In contrast, ChunkStream results are fairly consistent because ChunkStream fetches a more constant number of chunks to reach any frame in the video.

4.2.2. High-speed playback

Fig. 15 shows the bandwidth consumed as an editor fast-forwards through a stream at 2-, 4-, 8-, and 16-times real time. We show HTTP Live Streaming for comparison, even though the client must download entire segments before fast forwarding and accordingly behaves as if it is playing at normal speed. The amount of bandwidth consumed by ChunkStream falls in proportion to the number of frames skipped.

No context. The falloff in bandwidth consumed is most dramatic when ChunkStream ignores context (context=none in Fig. 15). However, ignoring context leads to distorted frames.

Full context. In contrast, fetching all context frames (context=full) shows a slow falloff for low playback speeds since the client must fetch almost all frames to fill in context. In *Elephants Dream*, I-frames are fetched at multiples of 12. At 40× play speed, we fetch frames at multiples of 4: 0, 4, 8, 12, and so on. When fetching full context, the client will download frames 0 through 7 so that it can fully decode frame 8. In the steady state, fetching full context causes the client to fetch 2/3 of the frames.

It is only when the playback speed is greater than the I-frame period of 12 frames, that we see a dramatic falloff in the amount of data transferred. This is because we can skip full "groups of pictures". For *Elephants Dream*, this happens at 16× playback speed. At 16X playback speed, the client with fetch frames at locations that are multiples of 16, which means that it will only download the I-frame for frame 0, the I-frame at frame 12 and 3 additional context frames for frame 16, the I-frame at frame 24 and 7 context frames for frame 32, before repeating the pattern at frame 48. In the steady state, the client fetches 1/4 of the frames, a healthy drop off.

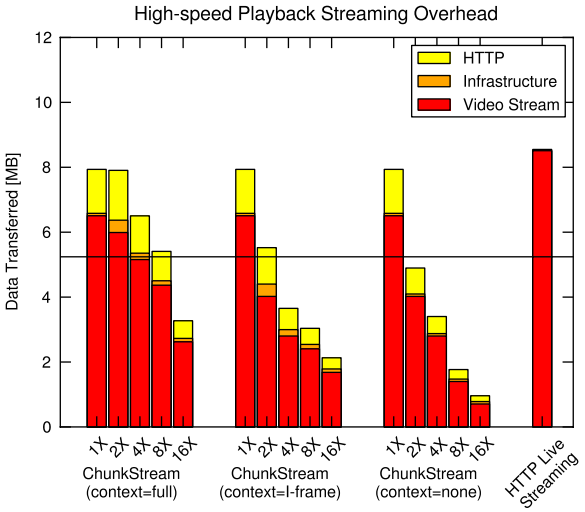

Fig. 15. Data transferred streaming a video at various play speeds. The horizontal line is the size of the raw video stream.

I-frames only. A reasonable compromise is to fetch only the I-frames in a context during high-speed playback, since I-frames contain the most information, can be decoded without distortion, and are found easily in a ChunkStream by following the first slot of a FrameContext chunk. Such a strategy has reasonable falloff as playback speed increases (shown by the context=I-frame bars), allowing editors to efficiently scan through long clips without having to download everything.

5. Related work

ChunkStream is inspired by existing work in video streaming protocols, analysis of user behavior for streaming systems, video editing, and network-enabled object systems.

5.1. Video streaming

Video streaming solutions fit into three general categories: streaming, downloading, and pseudo-streaming [9].

Two common streaming solutions are IETF's RTP/RTSP suite [10,11] and Adobe's RTMP [12]. RTP/RTSP uses a combination of specially "hinted" files, TCP control streams, and UDP data streams to give users frame-accurate playback over both live and pre-recorded streams. RTP/RTSP offers no way to modify videos in place. Adobe's RTMP [12] is a specialized protocol for streaming Flash-based audio and video over TCP. Unlike RTP/RTSP, RTMP multiplexes a single TCP connection, allowing it to more easily pass through firewalls and NATs at the cost of the complexity of multiplexing and prioritizing substreams within the TCP connection. Clients have some control over the stream and may seek to particular timestamps.

On the other hand, many clients "stream" video by downloading a complete video file over HTTP. HTTP downloading is simple, but does not allow efficient seeking in content that has not yet been downloaded, nor can it adapt to varying bandwidth constraints.

Pseudo-streaming is an extension of the download model that allows clients to view content as it is downloaded. Recently, Pantos' work on HTTP Live Streaming [3] extends the HTTP pseudo-streaming model by chunking long streams into smaller files that are individually downloaded. A streaming client first downloads a "playlist" that contains the list of video segments and how long each segment lasts. The client then streams each segment individually. The HTTP Live Streaming playlist may contain "variant" streams, allowing a client to switch to a, e.g., lower bandwidth, stream as conditions warrant. Seeking is still awkward, as the client must download an entire segment before seeking is possible.

Of the three categories, streaming is the most bandwidth efficient, but pseudo-streaming and downloading are the most widely implemented, likely because of implementation simplicity [9].

5.2. Streaming architectures

Xiao [13] surveys architectures for large-scale IP-based streaming services. Our current ChunkStream implementation is a traditional client/server architecture and as such, may not scale well to Internet-sized audiences. However, since chunks are small and individually addressable, future ChunkStream implementations may be able to make use of protocols and ideas from "mesh-pull" peer-to-peer video streaming systems like CoolStreaming/DONet [14] or PPLive [15] to scale video playback (if not editing) by allowing peers to host and share chunks.

5.3. Client interactivity

Costa et al. [1] studied four streaming workloads and found that almost all video clients start at the beginning of the clip and have a single interaction with the video: pause. All of the protocols mentioned above are optimized for the case where clients "tune in" to a stream and watch it without jumping to other parts of the video. In fact, most scalable streaming formats fail to scale when clients are interactive [2]. This is a problem for video editing because the editing process is highly interactive.

5.4. Video editing

Professional video editors use heavy-weight software like Avid [16], Adobe Premiere [17], or Apple Final Cut Pro [18] running on relatively well-equipped workstations. All three programs assume that media is accessible through some file system, scaling from local disks to networked petabyte storage arrays. Most consumers use simpler software like iMovie [19] to mix and mashup videos.

The Leitch BrowseCutter [20] is perhaps the first remote non-linear editing system. BrowseCutter allows users to work on laptops completely disconnected from the main store of high-quality video. In order to do so, each user downloads complete, highly compressed sets of low-quality video to their local machine and use standard editing tools to generate an edit decision list of cut points for all of the relevant clips in the final video. When the user is done, the user sends the EDL back to the server, which applies the edits to the high-quality video. ChunkStream borrows BrowseCutter's EDL concept, but gives the client complete control over what data to download, cache, and manipulate.

Sites like JayCut [21] enable simple video editing within a web browser. JayCut's video editor is an Adobe Flash application that presents a multi-track timeline and allows users to place transitions between videos in each track. JayCut does not offer a real-time preview of edits—if the user wants to view a series of edits, she must wait for JayCut to construct, transcode, and stream a brand new Flash video. In contrast, ChunkStream lets clients immediately review new content.

There is already a market for video editing applications on smartphones. Apple's iPhone 3GS includes a simple video editing application that can trim videos. Nexvio's ReelDirector [22] iPhone application provides a more comprehensive video editor with some special effects. In both cases, the video clips to be edited must be present on the user's iPhone.

5.5. Chunks

Chunks share properties with network-enabled persistent programming systems [23–25]. Persistent programming systems provide a hybrid programming model that offers the persistence of storage systems, but the computation model of object-oriented programming languages. Our chunk model differs from the persistent programming model in that chunks place an explicit limit on the amount of data a single chunk can hold, forcing us to make more granular data structures. We believe that starting with fine-grained data structures forces us to create data structures that are more accessible to small clients.

6. Future work

The architecture described in Section 2.2 uses a single namespace of chunks—the server assigns each chunk an identifier and these identifiers are the "locators" used inside chunk links. The identifiers used in the namespace must fit within a single slot, typically 32 or 64 bytes long, but are otherwise unconstrained.

Using a single namespace has several advantages, the two chief advantages being that (1) the server's namespace is globally resolvable by all of the clients that connect to the server, allowing clients to share links between themselves and (2) the server can control the namespace in such a way so as to prevent conflicts among clients. ChunkStream's single namespace is not limited to a single server – the namespace may span multiple servers within the same cluster or administrative domain as long as the servers agree on a chunk identifier allocation scheme – allowing a ChunkStream "server" to grow to an Internet-scale services like YouTube or Vimeo without changing the chunk interface or namespace.

A primary disadvantage of using the server's namespace is that clients must ask the server to allocate an identifier for any chunk they plan to use in ChunkStream. Editors using mobile devices in disconnected or poorly connected environments may not be able to create new EDL chunks even though their clients may have cached all of the relevant videos. A second disadvantage of using a single server namespace is that clients are limited to using only a single ChunkStream service at a time. This disadvantage is potentially more restrictive because it requires that all of the clips that one editor may want to use in his own clips reside on the same server. While it is possible to copy clips between servers, doing so takes time (especially for large videos copied across consumer-level Internet connections) and wastes bandwidth.

We are currently working on refactoring our ChunkStream implementation to allow peer-to-peer relationships among what are now ChunkStream servers and clients. The refactoring decomposes into two parts. First, we are working on a new naming scheme called *relative naming* that allows peers to create chunk identifiers for private chunks and then negotiate which shared names to use when a peer decides to share a heretofore private chunk. Relative naming allows many new features: an editor may create a mashup of videos from different servers as well as allow the editor's client to ingest new videos and create EDLs while disconnected. Second, we are working on a scheme to federate ChunkStream services that allows ChunkStream servers to retrieve ChunkStream clips from ChunkStream servers in other administrative domains,

without requiring the client to download the clips from one server and upload it to another. Federation further offloads work from clients by giving servers the ability to fetch clips for them.

7. Conclusions

Rather than using an ad hoc protocol and data format specially designed for video editing, ChunkStream uses generic chunks as a foundation for building larger data structures that specifically address the requirements of video editing. Chunks are of fixed size, offering many of the implementation advantages enjoyed by blocked streaming protocols while exposing the video-specific structure ignored by conventional streaming.

Although this paper uses video editing as a driving example, using chunks to build application-specific data structures is a generalizable technique. We believe that chunks may allow interchange of streamed data as well as open up data structures to more applications. Our future work will focus on other applications and deriving general principles for using chunks in network-enabled applications.

Acknowledgements

This work is sponsored by the T-Party Project, a joint research program between MIT and Quanta Computer Inc., Taiwan. The authors would like to thank Tim Shepard, the anonymous PerCom 2010 reviewers, and the anonymous PMC special issue reviewers for their comments on the paper and the approach ChunkStream takes.

References

[1] C.P. Costa, I.S. Cunha, A. Borges, C.V. Ramos, M.M. Rocha, J.M. Almeida, B. Ribeiro-Neto, Analyzing client interactivity in streaming media, in: WWW, 2004.
[2] M. Rocha, M. Maia, Ítalo Cunha, J. Almeida, S. Campos, Scalable media streaming to interactive users, in: ACM Multimedia, 2005.
[3] R. Pantos, HTTP live streaming, IETF, Internet Draft draft-pantos-http-live-streaming-01, Jun. 2009.
[4] Advanced video coding for generic audiovisual services, International Telecommunication Union, Recommendation H.264, May 2003. [Online]. Available: http://www.itu.int/rec/T-REC-H.264.
[5] G. Blakowski, R. Steinmetz, A media synchronization survey: reference model, specification, and case studies, IEEE Journal on Selected Areas in Communications 14 (1) (1996) 5–35.
[6] FFmpeg. [Online]. Available: http://ffmpeg.org/.
[7] B. Kurdali, Elephants dream, 2006. [Online]. Available: http://orange.blender.org/.
[8] M.A. Eriksen, Trickle: A userland bandwidth shaper for unix-like systems, in: FREENIX, 2005.
[9] L. Guo, S. Chen, Z. Xiao, X. Zhang, Analysis of multimedia workloads with implications for internet streaming, in: WWW, 2005.
[10] H. Schulzrinne, S. Casner, R. Frederick, V. Jackson, RTP: a transport protocol for real-time applications, IETF, RFC 1889, Jan., 1996.
[11] H. Schulzrinne, A. Rao, R. Lanphier, Real time streaming protocol, RTSP, IETF, RFC 2326, Apr., 1998.
[12] RTMP specification, Adobe Systems Inc., Adobe Developer Connection, Jan. 2009. [Online]. Available: http://www.adobe.com/devnet/rtmp/.
[13] Z. Xiao, F. Ye, New insights on internet streaming and IPTV, in: ACM ICCBIR, 2008.
[14] X. Zhang, J. Liu, B. Li, T.S.P. Yum, CoolStreaming/DONet: a data-driven overlay network for efficient live media streaming, in: IEEE INFOCOM, vol. 3, 2005.
[15] X. Hei, C. Liang, J. Liang, Y. Liu, K.W. Ross, A measurement study of a large-scale P2P IPTV system, IEEE Transactions on Multimedia 9 (8) (2007).
[16] Avid media composer software. [Online]. Available: http://www.avid.com/products/Media-Composer-Software/index.asp.
[17] Adobe premiere. [Online]. Available: http://www.adobe.com/products/premiere/.
[18] Final cut pro. [Online]. Available: http://www.apple.com/finalcutstudio/finalcutpro/.
[19] iMovie. [Online]. Available: http://www.apple.com/ilife/imovie/.
[20] R.S. Rowe, Remote non-linear video editing, SMPTE Journal 109 (1) (2000) 23–25.
[21] JayCut — online video editing. [Online]. Available: http://jaycut.com.
[22] Nexvio ReelDirector. [Online]. Available: http://www.nexvio.com/product/ReelDirector.aspx.
[23] V. Cahill, P. Nixon, B. Tangney, F. Rabhi, Object models for distributed or persistent programming, The Computer Journal 40 (8) (1997) 513–527.
[24] J.E.B. Moss, Design of the mneme persistent object store, ACM Transactions on Information Systems 8 (2) (1990) 103–139.
[25] C. Tang, D. Chen, S. Dwarkadas, M. Scott, Integrating remote invocation and distributed shared state, in: IPDPS, 2004.