Interactive Streaming of Structured Data

Justin Mazzola Paluska and Hubert Pham
MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, U.S.A.

***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.

I. 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 anemic compute 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.

While our exploration focuses on video, we believe that our approach is generalizeable to any structured data type, including other kinds of user data and executable programs.

A. 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 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.

B. 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

978-1-4244-5328-3/09/$25.00 ©2009 IEEE

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 protocols 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.

C. 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. ChunkStream exposes video clips as a series of chunks embedded in search trees that allow a client to quickly find frames, make edit decisions, and stream video.

II. 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.

A. Chunks

A chunk is a typed, ordered, fixed-sized array of fixed-sized slots. Each slot may be empty, contain scalar data, or hold a reference ("link") to another chunk. 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. Figure 1 illustrates a chunk that holds some text and links to another chunk that contains image data.

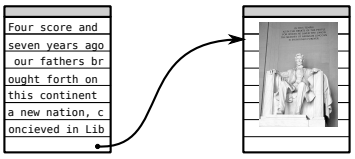

Figure 1. Two chunks with a link between them. The gray bar indicates the start of the chunk.

We chose chunks as our foundation data type for two reasons. First, since chunks contain a fixed number of fixed-sized 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. For example, 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. At the same time, since the chunk interface is fixed between client and server, chunk-based data structures can be shared between the two without content format mismatches or undefined links.

B. 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. Special effects, like a fade between two streams, is just a splice from the first stream to a new "special effect stream" to the destination stream and as such can be linked into our final video in the same way that any other stream could be.

1) Video Streams: We represent video streams with four types of chunks, as illustrated in Figure 2. The underlying stream is represented as a doubly-linked list of "backbone" chunks. The backbone is doubly-linked to allow forwards and backwards movement within the stream. For each frame in the video clip, the backbone links to a "LaneMarker" chunk that represents the frame.

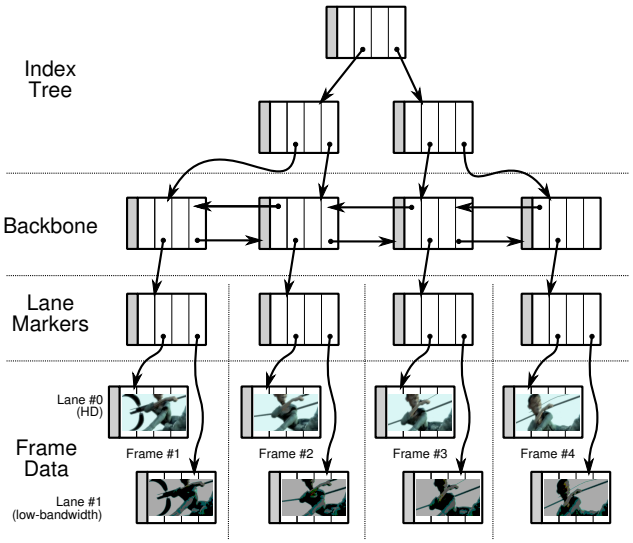

Figure 2. Chunk-based video stream representation used by ChunkStream. LaneMarker chunks represent logical frames and point to "lanes" composed of FrameData chunks. Each lane represents the frame in different formats, e.g. using HD or low-bandwidth encodings.

A LaneMarker chunk serves two purposes. First, it links to metadata about a frame, such as frame number or video time code. Second, the LaneMarker links to "lanes" of video. A lane is a video sub-stream 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 whatever lane it deems appropriate based on its resources.

Each lane slot in the LaneMarker points to a FrameData chunk that contains the underlying encoded video data. 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.

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 to the LaneMarker for the first frame and examines its contents to determine which lane to play. Finally, the client dereferences 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):
2
3       frame_num = 0
4       # Search through the IndexTree:
5       backbone = indextree_find_frame(tree_root, frame_num)
6
7       while playing:
8           # Dereference links from the backbone down to frame data:
9           lm_chunk = get_lane_marker(backbone, frame_num)
10          lane_num = choose_lane(lm_chunk)
11          frame_data = read_lane(lm_chunk, lane_num)
12          # Decode the data we have
13          deocde_frame(frame_data)
14
15          # Advance to next frame in backbone:
16          (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) 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 start frame of a clip to play and the length the clip should play; the order of the EDLClips in the linked list indicate 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 removed by changing the linked list. Using EDLs also allows ChunkStream to make video clips themselves immutable, aiding optimization and caching (cf. Section II-E).

Figure 3 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, then clip 2 for 96 frames, and then clip 1 again for 144 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.

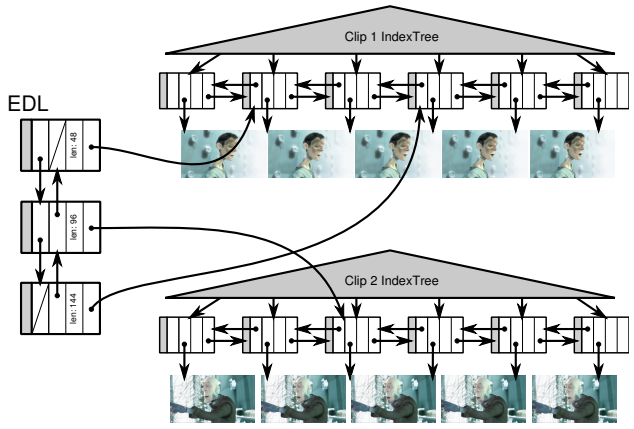

Figure 3. An Edit Decision List (EDL) using three portions of two clips.

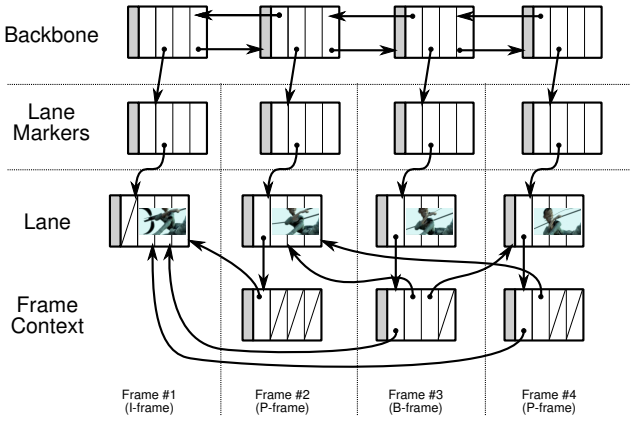

Figure 4. FrameContext chunks allow inter-frame compression by exposing the dependencies of a particular frame. The first slot in the FrameContext chunk always points to the I-frame that serves as the foundation of the dependent group of pictures and subsequent slots point to frames in the dependency chain between the I-frame and the current frame in the required decode order.

C. 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 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, usually provided by a search feature on the server.

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 lives on the server, allowing the user to switch clients and still access his work.

After running through a few iterations, the user is happy with his 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 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.

D. 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 Lane Markers 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 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-frame; and bi-predictive frames (B-frames) that depend on not only the previously displayed I- or P-frame, but also the *next* I- or P-frame to be displayed. 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. Rather than complicate the backbone, we augment the FrameData chunks that make up each lane with FrameContext chunks that specify the dependencies of each frame.

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 Figure 4, 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.

[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.

E. 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 explosion: (1) densely packed infrastructure chunks, (2) caches, and (3) server-side path de-referencing.

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 overhead. 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) 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. Moreover, 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 EDL or EDLClip chunks, since they are volatile.

3) Server-side Path De-referencing: In lines 9 and 10 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 generic 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. Figure 5 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 dereference the slot 1 in chunk A, and then dereference 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.

III. Implementation

In order to test our ChunkStream ideas, we built a prototype chunk server and a chunk client library. All source

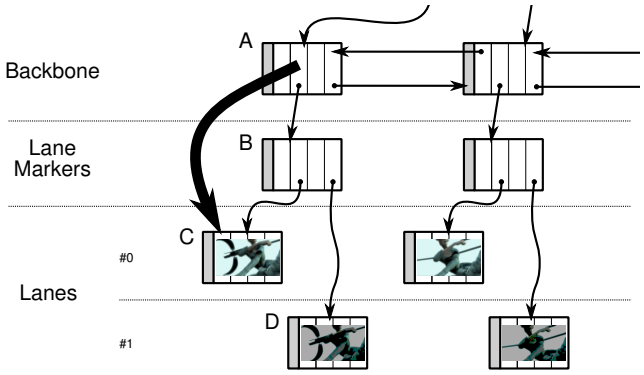

Figure 5. Clients using server-side path dereferencing may skip over LaneMarkers if they know what lane they want to use.

code is available under a free license from http://o2s.csail.mit.edu/chunks.

A. Server

The server exposes chunks over HTTP and may run in any web server that supports CGI and Python. Our server uses a simple numbering scheme to name chunks both over the network and within chunk links. The HTTP interface exposes a simple chunk protocol that 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 server-side path de-referencing. 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 Figure 5 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. However, read-only stores that simply expose a set of pre-computed chunks could be equally well-served by a standard web server serving static chunks without needing our CGI server script.

B. Client Library

Our client code includes utilities for transcoding and creating ChunkStreams from standard H.264 video clips as well as libraries to aid navigation and playback from a Chunk-based server. The client playback library also implements our server-side path de-referencing extension and includes an optional chunk cache.

IV. 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.

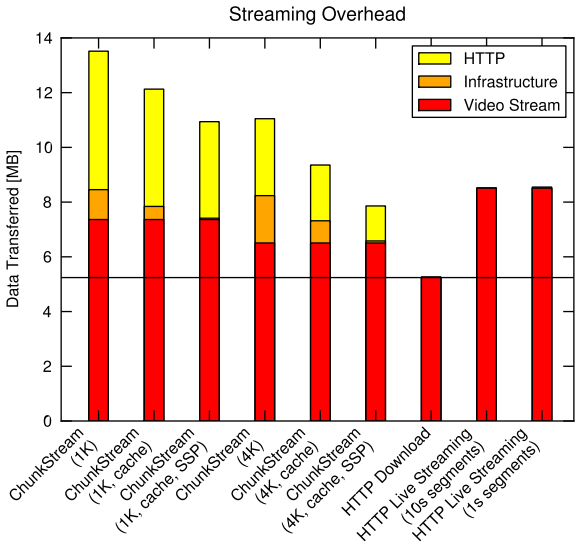

Figure 6. Data transferred streaming a video using ChunkStream, standard downloading, and HTTP Live Streaming. The horizontal line is the size of the raw video stream.

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.

For all of our tests, we used the first minute of the 720x405 24 frame/second H.264-encoded version of *Elephants Dream* [5] 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.

A. 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.

Figure 6 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 Figure 6, 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.

B. 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.

1) Frame-accurate Seeking: Figure 7 shows the amount of time it takes to seek to a specified frame in a the video clip using a cold cache. To mirror the network environment small devices may encounter, we use trickle [6] to simulate the optimal 2Mbit/s bandwidth of wireless 3G networks.

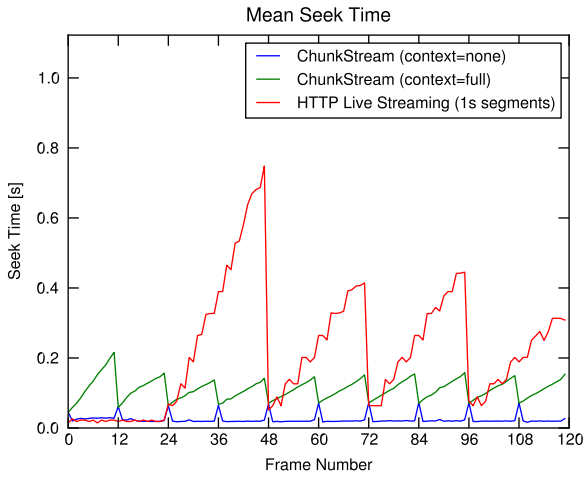

Figure 7. Mean seek time to seek to a random position in the stream. The clients are on a network limited to wireless 3G-like speeds of 2Mbit/s.

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, since those can be quickly downloaded. We do not show HTTP Download as that requires downloading the entire file, which takes many minutes.

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 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.

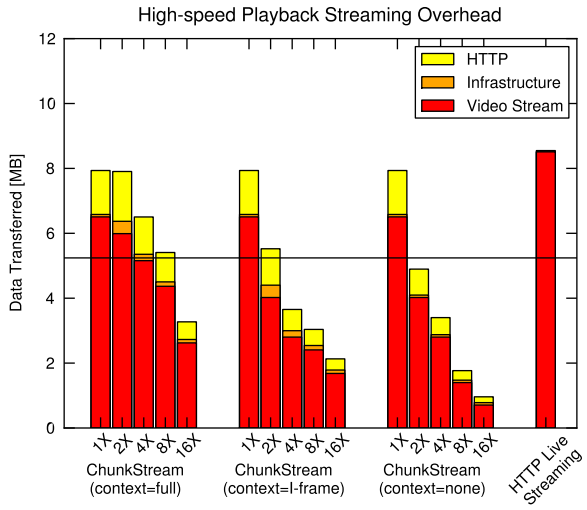

Figure 8. Data transferred streaming a video at various play speeds. The horizontal line is the size of the raw video stream.

2) High-speed Playback: Figure 8 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 it is playing as normal speed. The amount of bandwidth consumed by ChunkStream falls in proportion to the number of frames skipped. The fall-off is most dramatic when ChunkStream ignores context (context=none in Figure 8). However, ignoring context leads to distorted frames. 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. When the playback speed is greater than the frequency of I-frames, we are able to skip I-frames. In our case, we see a steep fall-off in bandwidth consumption at the 16X playback speed, the first playback step greater than the I-frame period of 12 frames.

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 fall-off as playback speed increases (shown by the context=I-frame bars), allowing editors to efficiently scan through long clips without having to download everything.

V. 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.

A. Video Streaming

Video streaming solutions fit into three general categories: streaming, downloading, and pseudo-streaming [7]. Two common streaming solutions are IETF's RTP/RTSP suite [8], [9] and Adobe's RTMP [10]. 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 [10] 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's 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 [7].

B. 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.

C. Video Editing

Professional video editors use heavy-weight software like Avid [11], Adobe Premiere [12], or Apple Final Cut Pro [13] 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 [14] to mix and mash up videos.

The Leitch BrowseCutter [15] 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 (EDL) 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 [16] 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 smart phones. Apple's iPhone 3GS includes a simple video editing application that can trim videos. Nexvio's ReelDirector [17] 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.

D. Chunks

Chunks share properties with network-enabled persistent programming systems [18], [19], [20]. 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.

VI. Conclusions and Future Work

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 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.

VII. 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 Steve Ward and Tim Shepard 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, and B. Ribeiro-Neto, "Analyzing client interactivity in streaming media," in *WWW*, 2004.

[2] M. Rocha, M. Maia, Ítalo Cunha, J. Almeida, and 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. http://www.itu.int/rec/T-REC-H.264

[5] B. Kurdali, "Elephants dream," 2006. http://orange.blender.org/

[6] M. A. Eriksen, "Trickle: A userland bandwidth shaper for unix-like systems," in *FREENIX*, 2005.

[7] L. Guo, S. Chen, Z. Xiao, and X. Zhang, "Analysis of multimedia workloads with implications for internet streaming," in *WWW*, 2005.

[8] H. Schulzrinne, S. Casner, R. Frederick, and V. Jackson, "RTP: a transport protocol for Real-Time applications," IETF, RFC 1889, 1996.

[9] H. Schulzrinne, A. Rao, and R. Lanphier, "Real time streaming protocol (RTSP)," IETF, RFC 2326, Apr. 1998.

[10] "RTMP specification," Adobe Systems Inc., Adobe Developer Connection, 2009. http://www.adobe.com/devnet/rtmp/

[11] "Avid media composer software." http://www.avid.com/products/Media-Composer-Software/index.asp

[12] "Adobe premiere." http://www.adobe.com/products/premiere/

[13] "Final cut pro." http://www.apple.com/finalcutstudio/finalcutpro/

[14] "iMovie." http://www.apple.com/ilife/imovie/

[15] R. S. Rowe, "Remote non-linear video editing," *SMPTE Journal*, vol. 109, no. 1, pp. 23—25, 2000.

[16] "JayCut - online video editing." http://jaycut.com

[17] "Nexvio ReelDirector." http://www.nexvio.com/product/ReelDirector.aspx

[18] V. Cahill, P. Nixon, B. Tangney, and F. Rabhi, "Object models for distributed or persistent programming," *The Computer Journal*, vol. 40, no. 8, pp. 513–527, Aug. 1997.

[19] J. E. B. Moss, "Design of the mneme persistent object store," *ACM Transactions on Information Systems*, vol. 8, no. 2, pp. 103–139, 1990.

[20] C. Tang, D. Chen, S. Dwarkadas, and M. Scott, "Integrating remote invocation and distributed shared state," in *IPDPS*, 2004.