Continuations for video decoding and scrubbing

Continuation Fest 2008, Tokyo

Conrad Parker

April 13, 2008

Continuations for video decoding and scrubbing

Playback of encoded video involves scheduling the decoding of audio and video frames and synchronizing their playback. "Scrubbing" is the ability to quickly seek to and display an arbitrary frame, and is a common user interface requirement for a video editor. The implementation of playback and scrubbing is complicated by data dependencies in compressed video formats, which require setup and manipulation of decoder state.

Conrad Parker

kfish on #haskell, and a PhD student at Kyoto University.

- **Type-Level Instant Insanity**, Monad.Reader #8
- **HOgg** Haskell Ogg library and tools
- **Sweep** sound editor (written in C, GTK+)
- **Annodex** video browsing and search

Outline of topics

- The goal
- The problems
- How continuations might be used to solve them

The eventual goal

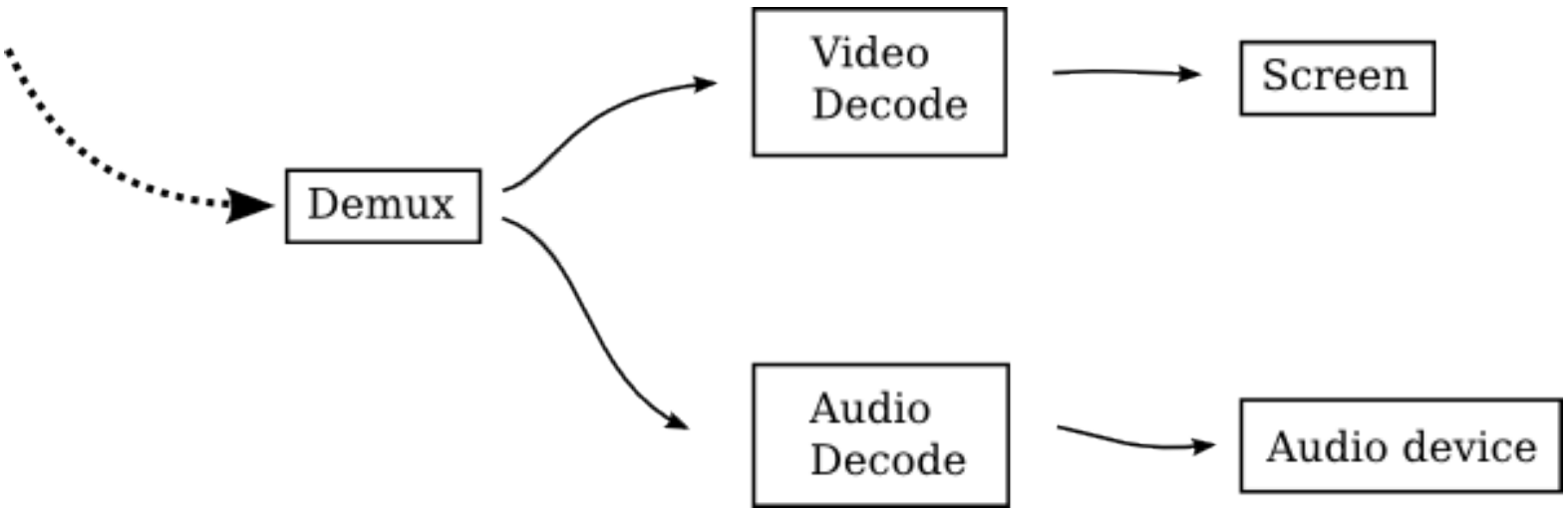
The basic operations of video playback are demux and audio+video decode, and the goal is to present these in sync, despite delays and drift.
Ultimately the available hardware dictates what is possible to decode. No amount of player re-architecture can reduce the math workload, but a carefully designed player architecture can ensure that the CPU use is scheduled among these tasks just-in-time, avoiding failure and minimizing memory use.

- BOSSA 2008, "Video player internals"

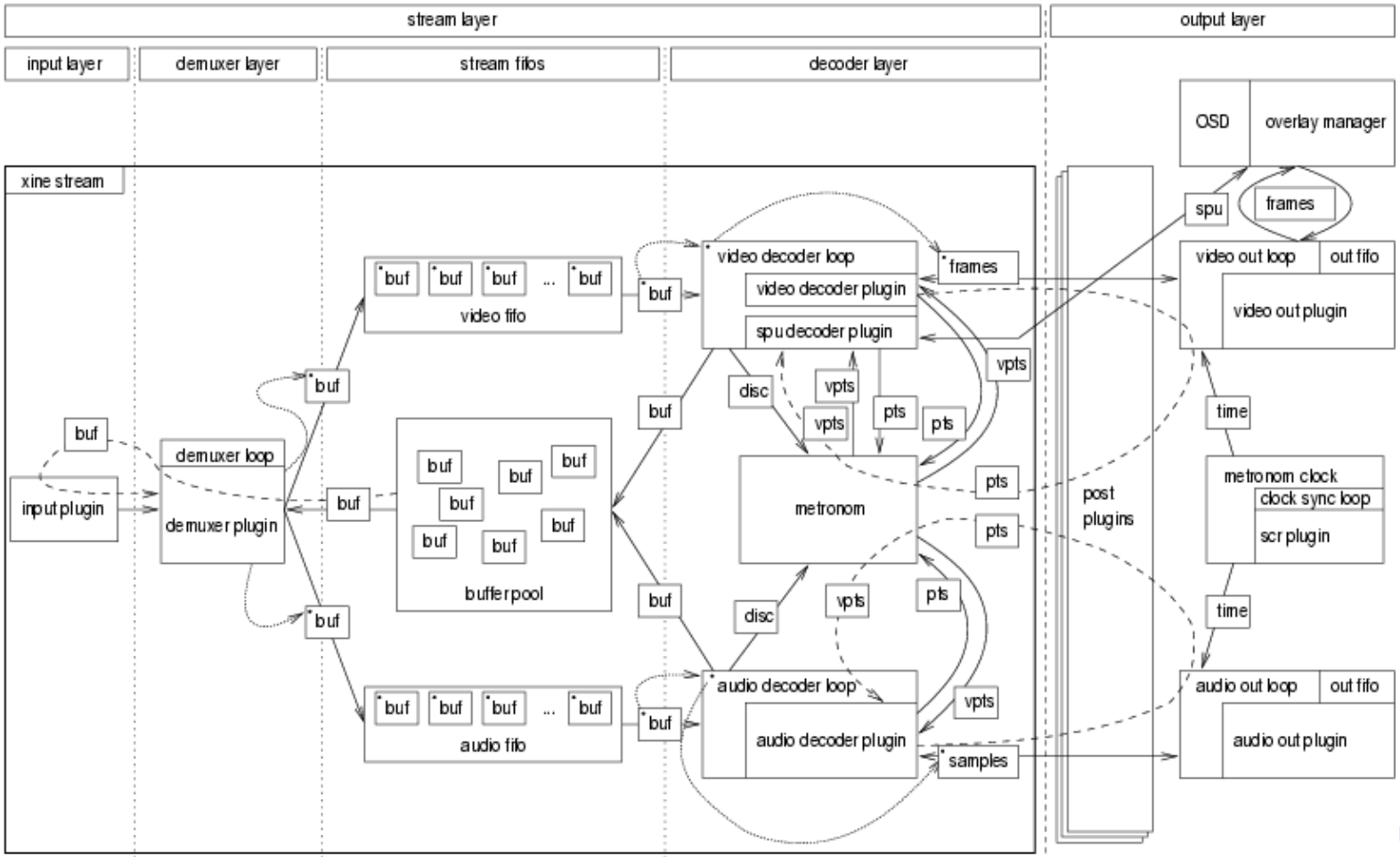
Traditional architecture

Most existing video players are built entirely statefully. Compressed video is data is read in, demuxed to stateful video and audio decoders, and immediately (via some buffers to absorb latency) rendered to the screen and speakers.

A simplified architecture diagram



The actual architecture diagram from xine



Subproblems

We will concentrate on two subproblems which are difficult to represent in a conventional player architecture:

- Seeking and scrubbing on compressed video
- Synchronization of audio/video decoders

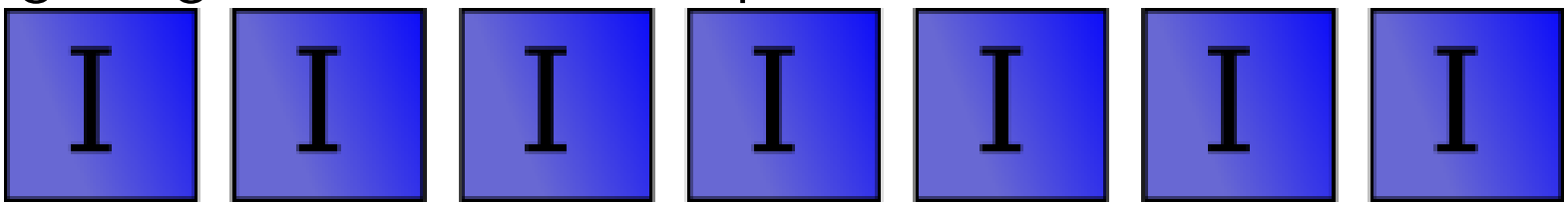
Subproblem: Seek and scrub

Let's consider problems which are inherent to the video track.

- Seeking and scrubbing on compressed video

Video frames

Ignoring audio, consider a sequence of video frames:



We would like some high-level operations on this `[Frame]`:

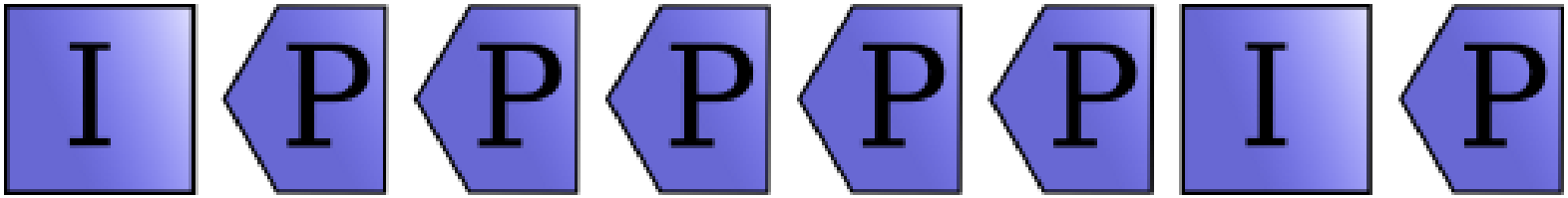
- next, prev frame
- seek to frame #

Anything else can be built out of these operations. We assume that rewind to beginning is just seek to frame 0, and we can implement seek to chapter if we have a table of contents.

Complications

- Video files are "large".
- Compressed frames are of varying sizes.
- Decoding a frame is CPU-intensive.
- Decoded video frames are very large.
- Due to frame dependencies, the decoder state must be carried between successive frames.

Frame dependencies: Predictive frames



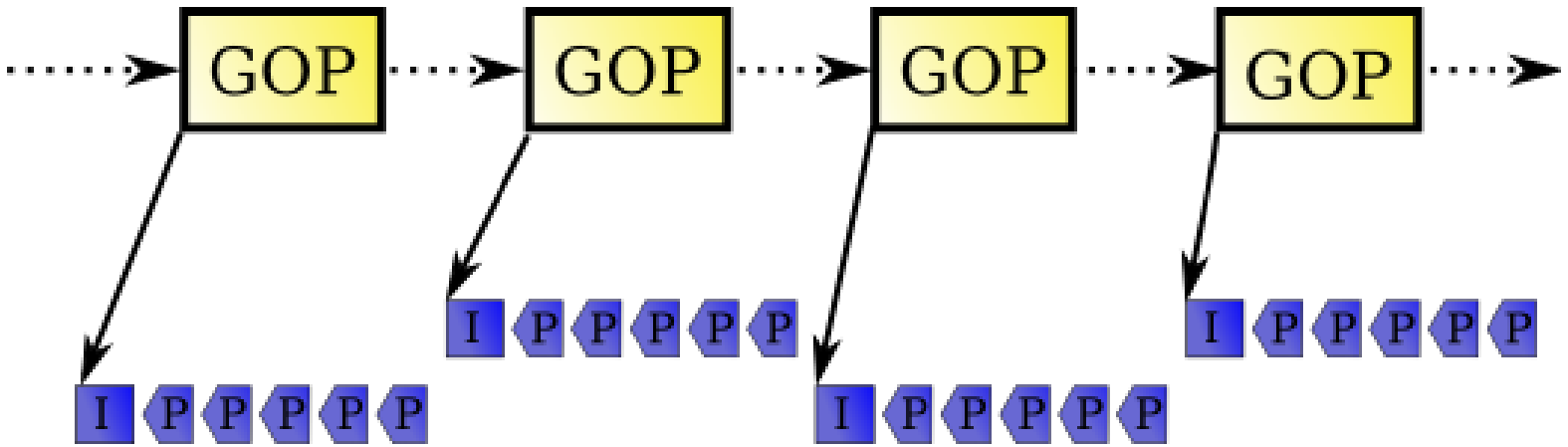
I: Intra-frame (Keyframe)
P: Predictive frame: only encode the difference from an earlier frame

- An Intra-frame and the Predictive frames which depend on it are collectively called a **group of pictures** (GOP).
- GOPs are independent of each other.
- This is the scheme used in Theora and MPEG-2 simple profile. More complex frame dependencies exist in other profiles and codecs, but they are still limited within a GOP.

Groups of Pictures (GOPs)

As GOPs are independent of each other, existing C decoder libraries simply reset the decode state when moving from one GOP to the next.

- Considering decode states, we can think of the video stream as a sequence of GOPs:



A zipper over GOPs and frames

This leads naturally to a zipper.
But, we want to evaluate the video frames lazily:

- We want to decode frames one at a time; we don't want to decode the whole video
- We want to be able to seek to an arbitrary point without decoding everything in-between, and without losing earlier decoded frames or earlier decoder states
- After we've seeked and started playing, we want to be able to discard frame data which is no longer needed.

Binary seeking on varaible bitrate codecs

Most modern codecs are variable bitrate, meaning that compressed frames are not a constant size; simple pictures can be encoded using few bits, and and the bits saved there are allocated to the encoding of more complex pictures. To find a given frame in a compressed file you either need a seek table, or you do a binary seek over the file. When doing such a binary seek, we can dynamically build a seek table for use in future seeks.

- The GOP type in the zipper always records its starting byte offset, even if it contains no decoded frames

Seeking over the network

It is also possible to perform the binary seek over the network. using **HTTP/1.1 byte range requests**:

- `Range: bytes=1000-2000`

Or **Annodex**.net timed URIs, which contain a time offset and return a new video:

- `http://media.example.com/index.axv?t=00:02:30`

Subproblem: Audio/video sync

Now let's consider complications that arise when we attempt to simultaneously decode and play audio which is interleaved with the compressed video data.

- Synchronization of audio/video decoders

Goal: Synchronize audio and video

The general aim is simply to present audio and video in sync.

- We say that audio and video are in sync when audio is heard at the same time as the appearance of the visual image representing it. Think of a clapper board in a film production.

Doing so involves receiving data from the network or disk, separating it into audio and video frames, interpreting and adjusting their timestamps, re-ordering and scheduling when to decode them, and finally rendering these to the respective devices, allowing for variability in rendering time.

Issue: Muxing can introduce data overruns

Blocks of audio and video have different durations, so they necessarily run ahead of each other. In a container like Ogg, many data packets from the same codec can be bundled together into the same Ogg page. Data overruns become a problem if, for example, we attempt to decode all available video data even though this is many frames ahead of the available audio data. Hence we waste CPU cycles and may not be able to render the audio in time, losing sync.

Measuring data overruns with oggplay-info

oggplay-info is a tool distributed with the liboggplay source. It uses oggplay's playback scheduler to measure data overruns in the encoded audio and video data.

```
Theora: Track 0
        Worst overrun: 45 frames
        Average overrun: 15.811 frames
        Histogram bucket size: 2.250
        Histogram: 5 10 10 9 12 10 9 5 6 4 4 3 3 2 2 2 3 2
        SD of overrun: 11.357817
```

Solution: Coroutines for audio/video decode

The decode states of the audio and video tracks are independent. Ideally, when one is given a block of encoded data (which may contain a variable number of encoded frames), we would like it to decode only up to a given duration, and then return. When it is next called, it should resume from exactly that point in the decode. This suggests that coroutines might be useful for managing the decoder states, to allow us to swap efficiently between audio and video decoding.

- ... rather than leaving it to naïve thread scheduling ...

Design: Continuations for video decoding and scrubbing

- A cursor into a stream of decoded video frames (a zipper over GOPs and frames).
- Frames are decoded lazily.
- Decoder states (per GOP) are recreated or restored when seeking.
- Replace a sequence of decoded frames by the continuation which created them when no longer required in memory.
- Schedule audio and video by coroutines to avoid problems caused by muxing overruns.

C Implementation issues

I'd like to implement this in C for embedded devices ...

- Currently playing with lazy lists, to make a zipper
- Lock-free implementation? (Not dependent on language runtime)
- Unfortunately, existing C codec libraries are very stateful.
- No `setcontext()`, `getcontext()` in uClibc.

Conclusion

Conventional architectures for video decoding are very stateful, which makes operations which affect all parts of a player very difficult to debug. Such operations include local and network seeking, and synchronizing the decode of audio and video. Perhaps continuation-based approaches can be useful to solve these problems.

- Conrad Parker `conrad@metadecks.org`