SVE: Distributed Video Processing at Facebook Scale

Qi Huang[1], Petchean Ang[1], Peter Knowles[1], Tomasz Nykiel[1], Iaroslav Tverdokhlib[1], Amit Yajurvedi[1], Paul Dapolito IV[1], Xifan Yan[1], Maxim Bykov[1], Chuen Liang[1], Mohit Talwar[1], Abhishek Mathur[1], Sachin Kulkarni[1], Matthew Burke[1,2,3], and Wyatt Lloyd[1,2,4]

[1]Facebook, Inc., [2]University of Southern California, [3]Cornell University, [4]Princeton University
qhuang@fb.com,wlloyd@princeton.edu

ABSTRACT

Videos are an increasingly utilized part of the experience of the billions of people that use Facebook. These videos must be uploaded and processed before they can be shared and downloaded. Uploading and processing videos at our scale, and across our many applications, brings three key requirements: low latency to support interactive applications; a flexible programming model for application developers that is simple to program, enables efficient processing, and improves reliability; and robustness to faults and overload. This paper describes the evolution from our initial monolithic encoding script (MES) system to our current Streaming Video Engine (SVE) that overcomes each of the challenges. SVE has been in production since the fall of 2015, provides lower latency than MES, supports many diverse video applications, and has proven to be reliable despite faults and overload.

1 INTRODUCTION

Video is a growing part of the experience of the billions of people who use the Internet. At Facebook, we envision a video-first world with video being used in many of our apps and services. On an average day, videos are viewed more than 8 billion times on Facebook [28]. Each of these videos needs to be uploaded, processed, shared, and then downloaded. This paper focuses on the requirements that our scale presents, and how we handle them for the uploading and processing stages of that pipeline.

SOSP '17, October 28, 2017, Shanghai, China

ACM ISBN 978-1-4503-5085-3/17/10.
https://doi.org/10.1145/3132747.3132775

Processing uploaded videos is a necessary step before they are made available for sharing. Processing includes validating the uploaded file follows a video format and then re-encoding the video into a variety of bitrates and formats. Multiple bitrates enable clients to be able to continuously stream videos at the highest sustainable quality under varying network conditions. Multiple formats enable support for diverse devices with varied client releases.

There are three major requirements for our video uploading and processing pipeline: provide low latency, be flexible enough to support many applications, and be robust to faults and overload. Uploading and processing are on the path between when a person uploads a video and when it is shared. Lower latency means users can share their content more quickly. Many apps and services include application-specific video operations, such as computer vision extraction and speech recognition. Flexibility allows us to address the ever increasing quantity and complexity of such operations. Failure is the norm at scale and overload is inevitable due to our highly variable workloads that include large peaks of activity that, by their viral nature, are unpredictable. We therefore want our systems to be reliable.

Our initial solution for uploading and processing videos centered around a monolithic encoding script (MES). The MES worked when videos were nascent but did not handle the requirements of low latency, flexibility, or robustness well as we scaled. To handle those requirements we developed the Streaming Video Engine (SVE), which has been in production since the fall of 2015.

SVE provides low latency by harnessing parallelism in three ways that MES did not. First, SVE overlaps the uploading and processing of videos. Second, SVE parallelizes the processing of videos by chunking videos into (essentially) smaller videos and processing each chunk separately in a large cluster of machines. Third, SVE parallelizes the storing of uploaded videos (with replication for fault tolerance) with processing it. Taken together, these improvements enable SVE to reduce the time between when an upload completes and a video can be shared by 2.3×−9.3× over MES.

SVE provides flexibility with a directed acyclic graph (DAG) programming model where application programmers write tasks that operate on a stream-of-tracks abstraction. The stream-of-tracks abstraction breaks a video down into tracks of video, audio, and metadata that simplify application programming. The DAG execution model easily allows programmers to chain tasks sequentially, while also enabling them to parallelize tasks, when it matters for latency.

Faults are inevitable in SVE due to the scale of the system, our incomplete control over the pipeline, and the diversity of uploads we receive. SVE handles component failure through a variety of replication techniques. SVE masks non-deterministic and machine-specific processing failures through retries. Processing failures that cannot be masked are simple to pinpoint in SVE due to automatic, fine-grained monitoring. SVE provides robustness to overload through a progressing set of reactions to increasing load. These reactions escalate from delaying non-latency-sensitive tasks, to rerouting tasks across datacenters, to time-shifting load by storing some uploaded videos on disk for later processing.

The primary contribution of this paper is the design of SVE, a parallel processing framework that specializes data ingestion (§4.1,§4.3), parallel processing (§4.2), the programming interface (§5), fault tolerance (§6), and overload control (§7) for videos at massive scale. In addition, the paper's contributions also include articulating the requirements for video uploading and processing at scale, describing the evolution from MES (§2.3) to SVE (§3), evaluating SVE in production (§4,§6.4,§7.3), and sharing experience from operating SVE in production for years (§8).

2 BACKGROUND

This section provides background on the full video pipeline, some production video applications, our initial monolithic encoding script solution, and why we built a new framework.

2.1 Full Video Pipeline

The *full video pipeline* takes videos from when they are captured by a person to when they are viewed by another. There are six steps: record, upload, process, store, share, stream. We describe the steps linearly here for clarity, but later discuss how and when they are overlapped.

Videos are uploaded after they are recorded. Either the original video or a smaller re-encoded version is uploaded, depending on a variety of factors discussed in Section 4.1. Video data is transferred from the user's device to front-end servers [27] within Facebook datacenters. There is high variance in upload times and some times are quite long. While the video is being uploaded, the front-end forwards it to the processing system (MES or SVE).

The processing system then validates and re-encodes the video. Validation includes verifying the video and repairing malformed videos. Verification ensures that an uploaded file is indeed a video as well as blocking malicious files, for example those that try to trigger a denial of service attack during encoding. Repairing malformed videos fixes metadata such as an incorrect frame count or a mis-alignment between the time-to-frame mappings in the video and audio tracks.

After the processing system validates a video it then re-encodes the video in a variety of bitrates. Re-encoding into different bitrates allows us to stream videos to user devices, with different network conditions, at the highest bitrates they can sustain. It also allows us to continuously adapt bitrates as the network conditions of individual users vary over time. Re-encoding is the most computationally intensive step.

Once the video is processed, we reliably store it in our binary large object (BLOB) storage system [6, 19] so that it remains available for future retrieval. Sharing a video is application specific, which we discuss below. Once a video is shared it can be streamed to the user's device.

This streaming is done by a video player on the user's device that downloads, buffers, and plays small chunks of the video from our CDN [13, 29]. The player can download each chunk in any bitrate with which we re-encoded the video. The player monitors network conditions and selects the highest bitrate that it can play back smoothly. Higher bitrates and less user-visible buffering are desirable, and encoding a video into multiple bitrates is necessary for this.

The focus of this paper is the *pre-sharing pipeline*, which is the part of the pipeline between when a video is recorded and when it can be shared. This includes the uploading, processing, and storing steps. Our goals for these steps are high reliability and low latency.

2.2 Production Video Applications

There are more than 15 applications registered with SVE that integrate video. The applications ingest over tens of millions of uploads per day and generate billions of tasks within SVE. Four well known examples are video posts, Messenger videos, Instagram stories, and Facebook 360. Video posts are created through Facebook's main interfaces and appear on a user's Timeline and in their friend's News Feeds along with a user post, thumbnails, and notifications to the target audience. This application has a complicated set of DAGs averaging 153 video processing tasks per upload. Messenger videos are sent to a specific friend or a group for almost real-time interaction. Due to a tight latency requirement, it has the simplest DAGs that average a little over 18 tasks per upload. Instagram stories are created by recording on Instagram and applying special filters through a stand-alone filter app interface. Its pipeline is slightly more complicated

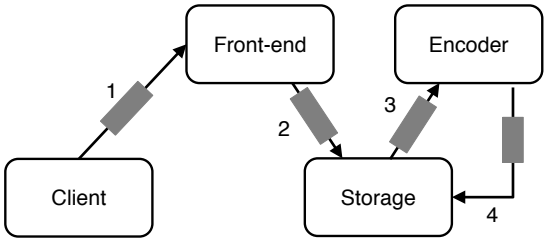


Figure 1: The initial video processing architecture. Videos were first uploaded by clients to storage via the front-end tier as opaque blobs (1–2). Then a monolithic encoder sequentially reencoded the video for storage (3–4) and ultimately sharing.

than the Messenger case, generating over 22 tasks per upload. 360 videos are recorded on 360 cameras and processed to be consumed by VR headsets or 360 players, whose pipeline generates thousands of tasks per upload due to high parallelism. Each application strives for low latency and high reliability for the best user experiences. Each application also has a diverse set of processing requirements, so creating reusable infrastructure at scale that makes it easy for developers to create and monitor video processing is helpful.

2.3 Monolithic Encoding Script

Our initial processing system was a Monolithic Encoding Script (MES). Figure 1 shows the architecture of the pre-sharing pipeline with MES. Under this design videos are treated as a single opaque file. A client uploads the video into storage via a front-end server. Once storage of the full video completes, the video is processed by an encoder within MES. The processing logic is all written in one large monolithic encoding script. The script runs through each of its processing tasks sequentially on a single server. There were many encoding servers, but only one handled a given video.

MES worked and was simple. But, it had three major drawbacks. First, it incurred high latency because of its sequential nature. Second, it was difficult to add new applications with appropriate monitoring into a single script. Third, it was prone to failure and fragile under overload. SVE addresses each of these drawbacks.

2.4 Why a New Framework?

Before designing SVE we examined existing parallel processing frameworks including batch processing systems and stream processing systems. Batch processing systems like MapReduce [10], Dryad [14], and Spark [35] all assume the data to be processed already exists and is accessible. This requires the data to be sequentially uploaded and then processed, which we show in Section 4.1 incurs high latency. Streaming processing systems like Storm [5], Spark Streaming [36], and StreamScope [16] overlap uploading and processing, but are designed for processing continuous queries instead of discrete events. As a result, they do not support our video-deferring overload control (§7), generating a specialized DAG for each video (§5), or the specialized scheduling policies we are exploring. This made both batch processing and stream processing systems a mismatch for the low-level model of computation we were targeting.

A bigger mismatch, however, was the generality of these systems. The interfaces, fault tolerance, overload control, and scheduling for each general system is necessarily generic. In contrast, we wanted a system with a video-specific interface that would make it easier to program. We also wanted a system with specialized fault tolerance, overload control, and scheduling that takes advantage of the unique characteristics of videos and our video workload. We found that almost none of our design choices—e.g., per-task priority scheduling (§3), a dynamically generated DAG per video (§5.2)—were provide by exisiting systems. The next four sections describe our video-specific design choices.

3 STREAMING VIDEO ENGINE

This section provides an overview of the architecture of SVE with the next four sections providing details. Section 4 explains how SVE provides low latency. Section 5 explains how SVE makes it simple to setup new processing pipelines. Sections 6 and 7 explain how SVE is robust to failure and overload, respectively.

Figure 2 shows a high-level overview of the SVE architecture. The servers involved in processing an individual video are shown. Each server type is replicated many times, though without failure only one front-end, preprocessor, and scheduler will handle a particular video.

There are four fundamental changes in the SVE architecture compared to the MES architecture. The first change is that the client breaks the video up into segments consisting of a group of pictures (GOP), when possible, before uploading the segments to the front-end. Each GOP in a video is separately encoded, so each can be decoded without referencing earlier GOPs. Segments split based on GOP alignment are, in essence, smaller stand alone videos. They can be played or processed independently of one another. This segmenting reduces latency by enabling processing to start earlier. The second change is that the front-end forwards video chunks to a preprocessor instead of directly into storage. The third change is replacing the MES with the preprocessor, scheduler, and workers of SVE. Disaggregating the encoding in these three components enables low latency through parallelization, higher reliability through fault isolation, and better reaction to overload through more options to shift

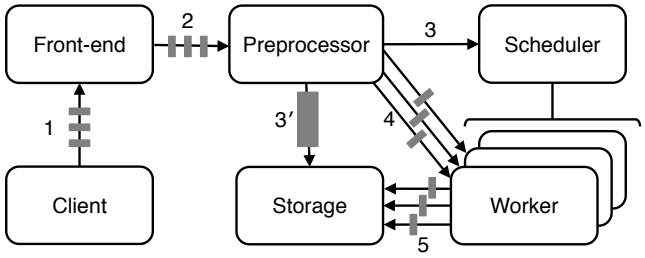


Figure 2: SVE architecture. Clients stream GOP split segments of the video that can be reencoded independently (1–2). The preprocessor does lightweight validation of video segments, further splits segments if needed, caches the segments in memory, and notifies the scheduler (3). In parallel, the preprocessor also stores the original upload to disk for fault tolerance (3′). The scheduler then schedules processing tasks on the workers that pull data from the preprocessor tier (4) and then store processed data to disk (5).

load. And the fourth change is that SVE exposes pipeline construction in a DAG interface that allows application developers to quickly iterate on the pipeline logic while providing advanced options for performance tunings.

Preprocessor. The preprocessor does lightweight preprocessing, initiates the heavyweight processing, and is a write-through cache for segments headed to storage. The preprocessing includes the validation done in MES that verifies the upload is a video and fixes malformed videos. Some older clients cannot split videos into GOP segments before uploading. The preprocessor does the GOP splitting for videos from those clients. GOP splitting is lightweight enough to be done on path on a single machine. It requires only a single pass over the timestamp for each frame within the video data, finding the nearest GOP boundaries to the desired segment duration to divide the video apart.

Encoding, unlike preprocessing, is heavyweight. It requires a pixel-level examination of each frame within a video track and operates both spatial image compression as well as temporal motion compression across a series of frames. Thus, video encoding is performed on many worker machines, orchestrated by a scheduler. The processing tasks that are run on workers are determined by a DAG of processing tasks generated by the preprocessor. The DAG that is generated depends on the uploading application (e.g., Facebook or Instagram) and its metadata, (e.g., normal or 360 video). We discuss how programmers specify the DAG in Section 5.

The preprocessor caches video segments after it preprocesses them. Concurrently with preprocessing, the preprocessor forwards the original uploaded segments to storage. Storing the segments ensures upload reliability despite the failures that are inevitable at scale. Caching the segments in the preprocessor moves the storage system, and its use of disks, off the critical path.

Scheduler. The scheduler receives the job to execute the DAG from the preprocessor as well as notifications of when different segments of the video have been preprocessed. The scheduler differentiates between low and high priority tasks, which are annotated by programmers when they specify the DAGs. Each cluster of workers has a high-priority and low-priority queue that workers pull tasks from after they finish an earlier task. The scheduler greedily places tasks from the DAG into the appropriate queue as soon as all of its dependencies are satisfied. Workers will pull from both queues when cluster utilization is low. When utilization is high they will only pull tasks from the high-priority queue. SVE shards the workload by video ids among many schedulers. At any time a DAG job is handled by a single scheduler.

Worker. Many worker machines process tasks in the DAG in parallel. Worker machines receive their tasks from the scheduler. They then pull the input data for that task either from cache in the preprocessor tier, or from the intermediate storage tier. Once they complete their task they push the data to intermediate storage if there are remaining tasks or to the storage tier if the DAG execution job has been completed.

Intermediate Storage. Different storage systems are used for storing the variety of data ingested and generated during the video encoding process. The choice of storage system depends on factors such as the read/write pattern, the data format, or access frequency for a given type of data. Metadata that is generated for use in the application is written to a multi-tier storage system that durably stores the data and makes it available in an in-memory cache [7] because the metadata is read much more frequently than it is written. The internal processing context for SVE is written to a storage system that provides durability without an in-memory cache. That context is typically written many times and read once, so caching would not be beneficial. The video and audio data is written to a specially configured version of our BLOB storage system. That storage system, and the one for internal processing context, automatically free this data after a few days because it is only needed while the video is being processed. We have found that letting this data expire instead of manually freeing it is both simpler and less error-prone: in case a DAG execution job gets stuck, which happens when a user halts an upload without an explicit signal, or when a DAG execution job state is lost due to system failure or bugs, letting the storage system manage freeing the data upper bounds the excess capacity without requiring explicit tracing or job recovery.

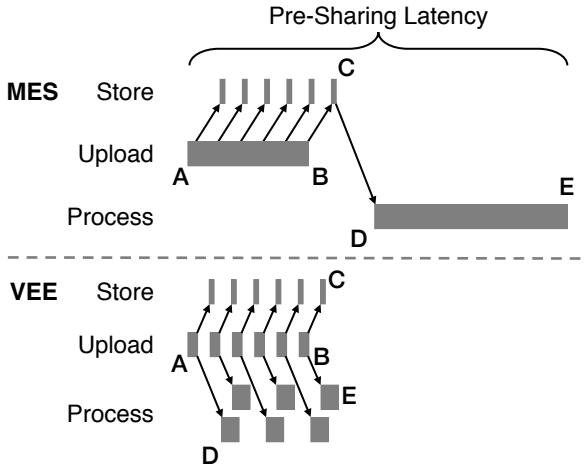


Figure 3: Logical diagrams of the pre-sharing latency of MES and SVE. Letters mark points in the diagram that are measured and reported in later figures.

4 LOW LATENCY PROCESSING

Low latency processing of videos makes applications more interactive. The earlier videos are processed, the sooner they can be shared over News Feed or sent over Messenger. This section describes how SVE provides low latency by overlapping uploading and processing (§4.1), processing the video in parallel (§4.2), and by overlapping fault tolerant storage and processing (§4.3). Figure 3 shows the logical pre-sharing latency for the MES and SVE designs to provide intuition for why these choices lead to lower latency. This section also quantifies the latency improvement that the SVE provides over MES (§4.4). All data is for News Feed uploads in a 6-day period in June 2017 unless otherwise specified.

4.1 Overlap Uploading and Encoding

The time required for a client to upload all segments of a video is a significant part of the pre-sharing latency. Figure 4a shows CDFs of upload times. Even for the smallest size class ($\leq$ 1 MB) approximately 10% of uploads take more than 10 seconds. For the 3–10 MB size class, the percentage of videos taking more than 10 seconds jumps to 50%. For the large size classes of 30–100 MB, 100–300 MB, 300 MB–1 GB, and $\geq$ 1 GB, more than half of the uploads take 1 minute, 3 minutes, 9 minutes, and 28 minutes, respectively. This demonstrates that upload time is a significant part of pre-sharing latency.

Uploads are typically bottlenecked by the bandwidth available to the client, which we cannot improve. This leaves us with two options for decreasing the effect of upload latency on pre-sharing latency: 1) upload less data, and 2) overlap uploading and encoding. One major challenge we overcome in SVE is enabling these options while still supporting the large and diverse set of clients devices that upload videos. Our insight is to opportunistically use client-side processing to enable faster sharing when it is possible and helpful, but to use cloud-side processing as a backup to cover all cases.

We decrease the latency for uploads through client-side re-encoding of the video to a smaller size when three conditions are met: the raw video is large, the network is bandwidth constrained, and the appropriate hardware and software support exists on the client device. We avoid re-encoding when a video is already appropriately sized or when the client has a high bandwidth connection because these uploads will already complete quickly. Thus, we prefer to avoid using client device resources (e.g., battery) since they will provide little benefit. Requiring all three conditions ensures we only do client-side re-encoding when it meaningfully decreases pre-sharing latency.

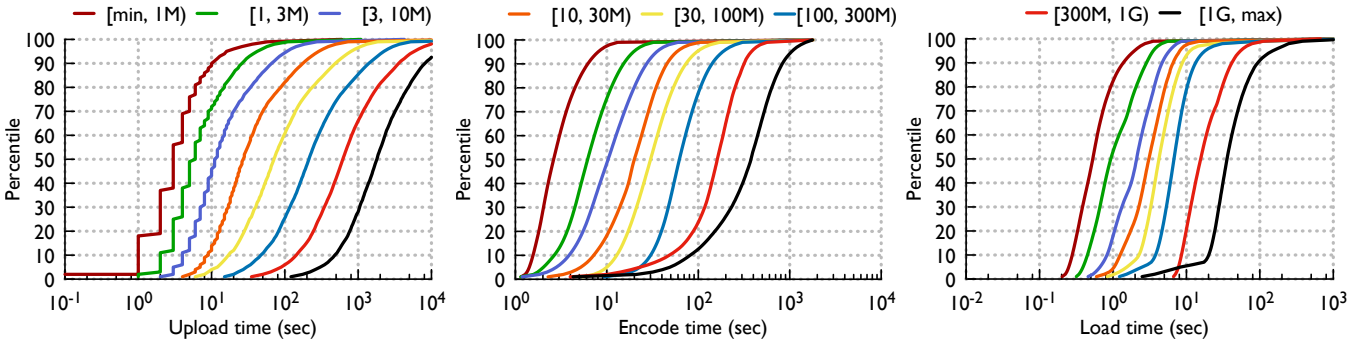
We decrease overall latency by overlapping uploading and server-side encoding so they can proceed mostly in parallel. This overlap is enabled by splitting videos into GOP-aligned segments. When there is client-side support for splitting, which is common, we do splitting there because it is a light-weight computation. When there is not client-side support, the preprocessor splits the video to enable parallelizing uploading and processing for all videos. As a result, the combined upload and processing latency can be as low as the upload latency plus the last segment processing latency.

4.2 Parallel Processing

The time required to process a video (D–E) is a significant part of the pre-sharing latency. Figure 4b shows CDFs of standard definition (SD) encoding time for different size classes of videos under MES. Unsurprisingly, there is a strong correlation between video size and encoding time. For the size classes smaller than 10 MB, most videos can be encoded in fewer than 10 seconds. Yet, for even the smallest size class, more than 2% of videos take 10 or more seconds to encode. For large videos the encoding time is even more significant: 53% of videos in the 100–300 MB size class take more than 1 minute, 13% of videos in the 300 MB–1 GB size class take more than 5 minutes, and 23% of videos larger than 1 GB take over 10 minutes. This demonstrates that processing time is a significant part of pre-sharing latency.

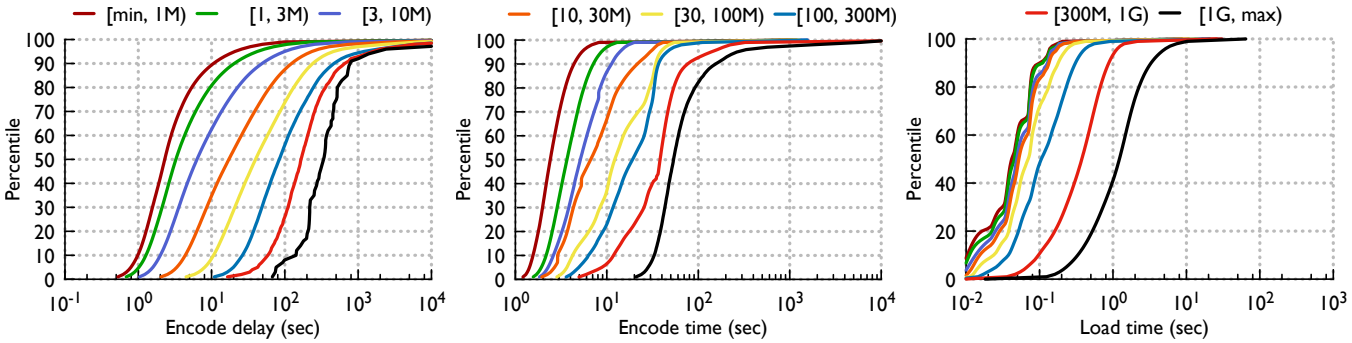
Fortunately, segmenting a video along GOP boundaries makes processing of the video parallelizable. Each segment can be processed separately from, and in parallel with, each other segment. The challenges here are in selecting a segment size, enabling per-segment encoding, and ensuring the resulting video is still well formed.

Segment size controls a tradeoff between the compression within each segment and parallelism across segments. Larger segments result in better compression because there



(a) Upload times (A–B). (b) MES encoding time (D–E). (c) MES load time (C–D).

Figure 4: Latency CDFs for the steps in the logical flow of MES broken down for ranges from < 1 MB to > 1 GB. The upload time (A–B) is the same for MES and SVE. Storage sync time (B–C) is described in §4.3. There is a single legend across the three figures.



(a) SVE encoding start delay (A–D). (b) SVE encoding time (D–E). (c) SVE per-video max fetch time.

Figure 5: Latency CDFs for the steps in the logical flow of SVE broken down for ranges from < 1 MB to > 1 GB. The upload time (A–B) is shown in Figure 4a. The storage time (B–C) is described in §4.3. Figure 5c shows the per-video max latency across all fetches by workers from the preprocessor caches (similar to C–D in MES). There is a single legend spread across the three figures.

is a larger window over which the compression algorithm can exploit temporal locality, but less parallelism because there are fewer segments. We use a segment size of 10 seconds for applications that prefer lower latency over the best compression ratio—e.g., messaging and the subset of encodings that drive News Feed notifications. We use a segment size of 2 minutes for high quality encodings of large video posts where a single digit percentage improvement on compression is prioritized, as long as the latency does not exceed a product-specified limit.

Per-segment encoding requires converting processing that executes over the entire video to execute on smaller video segments. SVE achieves this by segmenting each video, with each segment appearing to be a complete video. For videos with constant frame rates and evenly distributed GOP boundaries, there is no need for additional coordination during encoding. But, for variable frame rate videos, SVE needs to adjust the encoding parameters for each segment based on the context of all earlier segments—e.g., their frame count and duration. Stitching the separately processed segments back together requires a sequential pass over the video, but fortunately this is lightweight and can be combined with a pass that ensures the resulting video is well formed.

The high degree of parallelism in SVE can sometimes lead to malformed videos when the original video has artifacts. For instance, some editing tools set the audio starting time to a negative value as a way to cut audio out, but our encoder behaves differently when processing the audio track alone than in the more typical case when it processes it together with the video track. Another example is missing frame information that causes our segmentation process to fail to generate the correct segment. Ensuring SVE can handle such

cases requires repair at the preprocessing and track joining stages that fixes misaligned timestamps between video and audio tracks and/or refills missing or incorrect video metadata such as frame information. We observe that 3% of video uploads need to be repaired, mainly at the preprocessing stage. 68% of these repairs are to fix framerates that are too low or variable, 30% of them are to fix incomplete or missing metadata, and the remaining 2% are due to segmentation issues. Repairing videos at the preprocessing stage prevents overlapping upload and processing for this fraction of videos. We are investigating parallelizing the repair process with the upload to reduce end-to-end latency for such uploads.

4.3 Rethinking the Video Sync

The time required to durably store an uploaded video is sometimes a significant part of the overall latency. This *syncing time* (B–C) includes ensuring the video has been synced to multiple disks. This time is independent of the size of the video. It has a median of 200 ms, a 90th percentile of 650 ms, and a 99th percentile of 900 ms. This demonstrates that the syncing time is significant for some videos.

To reduce the effect of the latency that comes from durably storing the video, we overlap it with processing the video. This is essentially rethinking the sync [23, 24] for storage from the MES design. The MES design stored the uploaded video before processing to ensure it would not be lost once uploaded. The SVE design stores the uploaded video in parallel with processing it and waits for both steps to complete before notifying the user. This decreases the pre-sharing latency with the same fault tolerance guarantees.

After syncing the video to storage the MES encoder loads the video from the storage system (C–D). Figure 4c shows CDFs of the time required to load videos. The 90th percentile of loading times for all size classes of videos is over 1.3 seconds. For most videos that are 100 MB or larger, the loading time is over 6 seconds. This demonstrates that the loading time is significant for some videos. To reduce the latency required to fetch video segments, SVE caches them in memory in the preprocessor. This enables workers to fetch the segments without involving the storage system or disks.

4.4 SVE Latency Improvements

Figure 5 roughly parallels Figure 4 to show the improvement in the corresponding part of the logical flow from MES to SVE. Figure 5a shows the delay from when a video upload starts until when SVE can start processing it (A–D). This significantly reduces latency compared to MES where a video needed to be fully uploaded (A–B), synced to storage (B–C), and then fetched from storage (C–D) before processing could begin. Figure 5c shows the per-video maximum latency

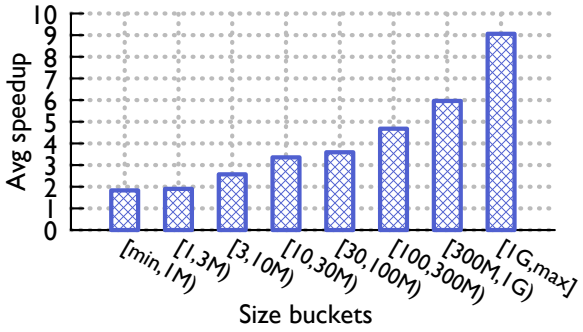


Figure 6: Speedup of SVE over MES in post-upload latency (B–E) broken down by video size.

for workers fetching segments from the preprocessor. Comparing this maximum fetch-from-cache time in SVE to the fetch-from-storage time in MES (C–D) shows SVE eliminates the tail of high latency for all videos. Figure 5b shows the encoding latency for SVE (D–E) targeting the same videos as MES encodes in Figure 4b with the same SD quality and 10 second segment size. Compared to the encoding times for MES, SVE delivers lower latency processing for all size classes, and especially for large videos.[1]

To get a better picture of SVE's improvement in latency over MES we tracked the post-upload latency (B–E) during our replacement of MES with SVE. Figure 6 shows the speedup that SVE provides over MES. The speedup ranges from 2× for ≤ 3 MB video to 9× for ≥ 1 GB videos. This demonstrates SVE provides much lower latency than MES.

5 DAG EXECUTION SYSTEM

There are an increasing number of applications at Facebook that process videos. Our primary goal for the abstraction that SVE presents is thus to make it as simple as possible to add video processing that harnesses parallelism (§5.1). In addition, we want an abstraction that enables experimentation with new processing (§5.2), allows programmers to provide hints to improve performance and reliability (§5.3), and that makes fine-grained monitoring automatic (§5.4). The stream-of-tracks abstraction achieves all of these goals.

5.1 DAGs on Streams-of-Tracks

The DAG on a stream-of-tracks abstraction helps programmers balance the competing goals of simplicity and parallelism. Programmers write processing tasks that execute sequentially over their inputs and then connect tasks into a DAG. The vertices of the DAG are the sequential processing tasks. The edges of the DAG are the dataflow. The

[1]MES appears to have lower tail latency for the largest size classes because it times out and aborts processing after 30 minutes. If it were to fully process those videos it would have much larger tail latencies than SVE.

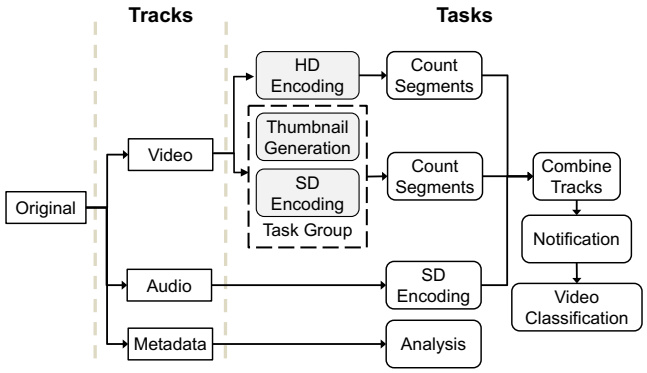


Figure 7: Simplified DAG for processing videos. Grayed tasks run for each segment of the video track.

```
pipeline = create_pipeline(video)

video_track = pipeline.create_video_track()
if video.should_encode_hd
  hd_video = video_track.add(hd_encoding)
   .add(count_segments)
sd_video = video_track.add(
  {sd_encoding, thumbnail_generation},
 ).add(count_segments)

audio_track = pipeline.create_audio_track()
sd_audio = audio_track.add(sd_encoding)

meta_track =
  pipeline.create_metadata_track()
  .add(analysis)

pipeline.sync_point(
  {hd_video, sd_video, sd_audio},
  combine_tracks,
 ).add(notify, 'latency_sensitive')
  .add(video_classification)
```

Figure 8: Pseudo-code for generating the simplified DAG. Dependencies in the DAG are encoded by chaining tasks. Branches of the DAG can be merged with sync points. Tasks can also be annotated easily, e.g., specifying the notify task to be latency sensitive.

granularity of both tasks and their inputs controls the complexity/parallelism tradeoff and is specified as a subset of a stream-of-tracks.

The stream-of-tracks abstraction provides two dimensions of granularity that reflect the structure of videos. The first dimension is the tracks within a video, e.g., the video track and the audio track. Tasks can operate on either one track individually or all tracks together. Specifying a task to operate on an individual track enables SVE to extract some parallelism and is simple for programmers. For example, speech recognition only requires the audio track while thumbnail extraction and facial recognition only require the video track. Specifying these tasks to operate on only their required track allows SVE to parallelize their execution without increasing the burden on the programmer because the processing tasks do not need to be rewritten.

The second dimension of granularity is the stream of data within a track, e.g., GOP-based segments within a video track. This dimension exposes more parallelism, but increases complexity because it requires tasks that can operate at the granularity of individual segments. For instance, enabling re-encoding tasks to operate on segments required us to modify the ffmpeg commands we used and required us to add a new task that stitches together the segmented video into a single video. Computer vision based video classification is an example of a task that it would be difficult to convert to operate at the segment level. Our classifier operates on the full video and does things like track objects across frames. Reengineering this classifier to operate across segments and then combine the different results would be complex.

Figure 7 shows a simplified version of the DAG for processing videos to be shared on Facebook. The initial video is split into tracks for video, audio, and metadata. The video and audio tracks are then copied n times, one for each of the n encoding bitrates ($n = 2$ for video, 1 for audio) in the figure. At this point, the re-encoding tasks, which are the most computationally intensive, are operating at the maximum parallelism: segments of individual tracks. Thumbnail generation, which is also moderately time consuming, is grouped inside the SD encoding task group to be executed at segment level, without incurring an additional video track copy. The output segments of each track are checked after they finish encoding in parallel, by the "count segments" tasks as a synchronization point. Then all the tracks are joined for storage, before the user is notified their video is ready to be shared. Some processing on the full video typically happens after the notification, such as video classification.

The DAG in Figure 7 follows the typical pattern of our DAGs: split into tracks, segment, split into encodings, collect segments, join segments, and then join tracks. This structure enables the most parallelism for the most computationally intensive tasks, which are re-encodings. It also provides a simple way for programmers to add most tasks. Most tasks operate over the fully joined tracks, which is even simpler to reason about than one big script. This provides SVE with most of the best of both worlds of parallelism and simplicity: parallelism is enabled for the few tasks that dominate processing time, which gives us most of the benefits of parallelism without requiring programmers to reason about parallelism for more than a few tasks.

5.2 Dynamic DAG Generation

The DAG for processing an individual video is dynamically generated at the beginning of that video's upload by the preprocessor. The preprocessor runs code associated with the uploading application to generate the DAG. For example, Figure 8 shows pseudo-code for generating the DAG shown in Figure 7.

Dynamic generation of the DAG enables us to tailor a DAG to each video and provides a flexible way to tune performance and roll out new features. The DAG is tailored to each video based on specific video characteristics forwarded from the client or probed by the preprocessor. For instance, the DAG for a video uploaded at a low bitrate would not include tasks for re-encoding that video at a higher bitrate. As another example, the width of parallelism for tasks operating on segments of video depends on the length of the video and the specified segment size. For instance, a 120-second video with a segment size of 10 seconds would be split into 12 segments. Dynamic generation makes it simple to tune performance through testing different encoding parameters, e.g., encode a small fraction of uploads in a different way to see if they result in smaller segments on average. It also makes it simple to roll out new features, e.g., enable a new feature only if the uploading user is an employee.

5.3 DAG Execution and Annotations

Once a DAG is generated, the preprocessor forwards it to the scheduler, which tracks the dependencies, dispatches tasks to workers, and monitors progress updates from the workers. The preprocessor regularly updates the scheduler with the readiness of each segment and each track for a given stream/DAG to enable the scheduler to dispatch tasks as soon as a segment becomes available.

All workers are equipped with HHVM [12], a virtual machine supporting just-in-time compilation for all SVE tasks, which in turn are Hack [31] functions deployed continuously from our code repository. During execution, each task is wrapped within framework logic that communicates with SVE components to prepare input, propagate output, and report task results.

Our DAG specification language has three types of annotations that programmers can add to tasks to control execution. The first annotation is a *task group*, which is a group of tasks that will be scheduled and run together on a single worker. By default, each task is an independent task group. Combining multiple task into a group amortizes scheduling overhead and eliminates cross-machine dataflow among these tasks. This provides the same performance as running all these tasks within a single task. But, it also allows for finer-grained monitoring, fault identification, and fault recovery.

Component	Strategy
Client device	Anticipate intermittent uploads
Front-end	Replicate state externally
Preprocessor	Replicate state externally
Scheduler	Synchronously replicate state externally
Worker	Replicate in time
Task	Many retries
Storage	Replicate on multiple disks

Figure 9: Fault tolerance strategies for SVE.

The second annotation is whether or not a task is *latency-sensitive*, i.e., a user is waiting for the task to finish. Tasks that are on path to alerting user that their video is ready to share are typically marked latency-sensitive. Tasks that are not are typically marked not latency-sensitive. Task groups with at least one latency-sensitive task are treated as latency-sensitive. The ancestors of a latency-sensitive task are also treated as latency-sensitive. Thus, in Figure 8 only the notification event needs to be marked latency-sensitive. These annotations are used by the scheduler for overload control.

5.4 Monitoring and Fault Identification

Separating processing out into tasks also improves monitoring and fault identification. On receiving a task group from scheduler, a worker executes its framework runtime to understand the incoming task composition, trigger tasks in order, and automatically add monitoring to every task. Monitoring helps us make sure that DAG jobs are succeeding or, if not, it helps us quickly identify the task that is failing.

Monitoring is also useful for analysis across many jobs. For instance, which tasks take the longest time to complete? Or, which tasks fail the most? We have found that monitoring tasks, identifying failing tasks, and doing this type of analysis is far easier with the SVE design than it was with MES.

6 FAULT TOLERANCE

Our scale, our incomplete control over the system, and the diversity of our inputs conspire to make faults inevitable. This section describes how SVE tolerates these failures.

6.1 Faults From Scale

SVE is a large scale system and, as with any system at scale, faults are not only inevitable but common. Figure 9 summarizes the fault tolerance strategies for components in SVE. We provide details on our strategy for handling task failure because we found it to be the most surprising.

A worker detects task failure either through an exception or a non-zero exit value. SVE allows programmers to configure the framework's response to failure according to

exception type. For non-recoverable exceptions such as the video being deleted due to user cancellation or site integrity revocation during an upload, SVE terminates the DAG execution job and marks it as canceled. Cancellation accounts for less than 0.1% of failed DAG execution jobs. (Most failed DAG execution jobs are due to corrupted uploads, e.g., those that do not contain a video stream.) In recoverable or unsure cases the worker retries the tasks up to 2 more times locally. If the task continues to fail, the worker will report failure to the scheduler. The scheduler will then reschedule the task on another worker, up to 6 more times. Thus, a single task can be executed as many as 21 times. If the last worker also fails to complete the task, then the task and its enclosing task group are marked as failed in the graph. If the task was necessary, this fails the entire DAG execution job.

We have found that this schedule of retries for failure helps mask and detect non-deterministic bugs. It masks non-deterministic bugs because multiple reexecutions at increasing granularities make it less and less likely we will trigger the bug. At the same time, our logs capture that reexecution was necessary and we can mine logs to find tasks with non-negligible retry rates due to non-deterministic bugs.

We have found that such a large number of retries does increase end-to-end reliability. Examining all video-processing tasks from a recent 1-day period shows that the success rate excluding non-recoverable exceptions on the first worker increases from 99.788% to 99.795% after 2 retries; and on different workers increases to 99.901% after 1 retry and 99.995% ultimately after 6 retries. Local retries have been particularly useful for masking failures when writing metadata and intermediate data, as the first local retry reduces such failures by over 50% (though second retry only further reduces failures by 0.2%). Retries on new workers have been effective for overcoming failures localized to a worker, such as filesystem failures, out-of-memory failures, and task specific processing timeouts triggered by CPU contention.

6.2 Faults from Incomplete Control

We do not have complete control over the pre-sharing pipeline because we do not control the client's device and its connectivity. The loss of a connection to a client for even a prolonged period of time (i.e., days) does not indicate an irrecoverable fault. As discussed in Section 4.1, some uploads take over a week to complete with many client-initiated retries after network interruptions.

A very slowly uploaded video takes a different path than a normal video and is protected with a grace period that is many days long. Segments from the slowly uploading videos will fall out of the preprocessor cache after a few hours. To protect against this we store the original segments from videos that have not been fully uploaded for the grace period. The scheduler will also purge the DAG execution job associated with the upload after a few days. To protect against this, if and when the upload does finish, SVE detects this special case and schedules the job for execution again. If, on the other hand, after the grace period the upload has not succeeded, then the preprocessed and processed segments are deleted.

6.3 Faults from Diverse Inputs

There are many different client devices that record and upload videos, and many different types of segments. On an average day, SVE processes videos from thousands of types of mobile devices, which cover hundreds of combinations of video and audio codecs. This diversity, combined with our scale, results in an extreme number of different code paths being executed. This in turn leads to bugs from corner cases that were not anticipated in our testing. Fortunately, our monitoring identifies these bugs so we can fix them later.

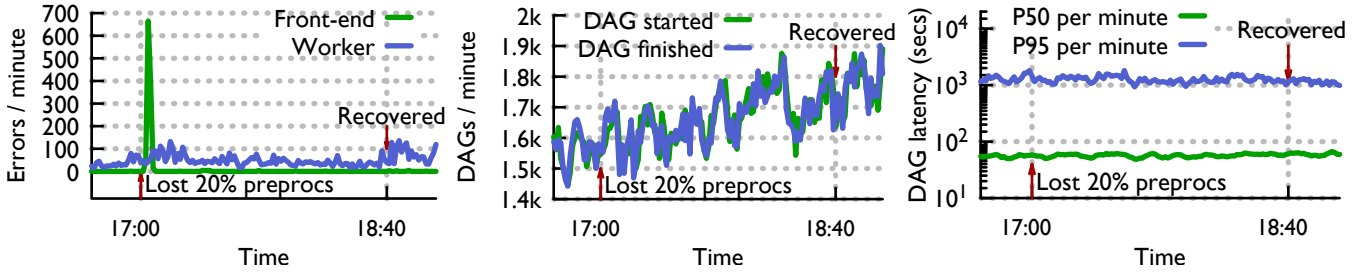
6.4 Fault Tolerance in Action

SVE tolerates small scale failures on a daily basis. To demonstrate resilience to large scale failures we show two failures scenarios with regional scope. In SVE, a *region* is a logical grouping of one or more data centers within a geographic region of the world. For the first scenario we inject failure for 20% of the preprocessors in a region. For the second scenario, which happened naturally in production, 5% of the workers in a region are gradually killed.

The first scenario is shown in Figure 10. We kill 20% of the preprocessors in a region at 17:00 and then bring them back at 18:40. Figure 10a shows the error rate at front-ends and workers in this region. There is a spike in the error rate at front-ends when we kill the preprocessors because the front-end-to-preprocessor connections fail. The impact of these failures is that those front-ends must reconnect the uploads that were going to the failed preprocessors to new preprocessors. Further, the clients may need to retransmit segments for which they will not get end-to-end acknowledgments.

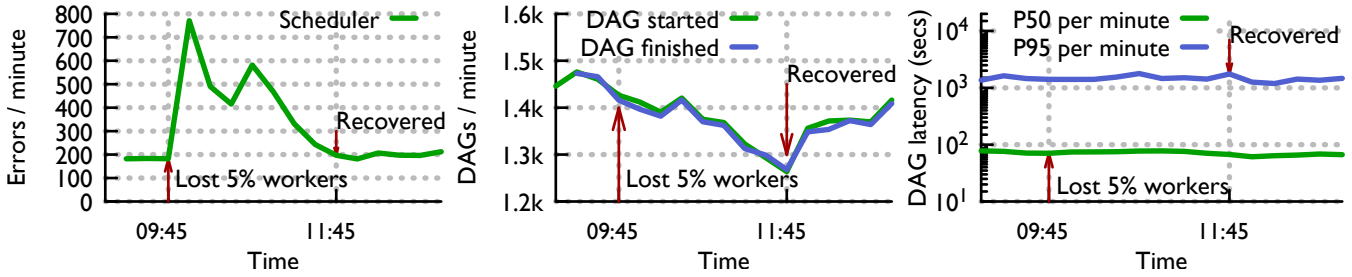
There is a small spike in error rate at the workers when the preprocessors are killed and when they recover. When the preprocessors are killed, ongoing segment fetches from them fail. Workers will then retry through a different preprocessor that will fetch the original segment from storage and re-preprocess it, if necessary. When the preprocessors recover, the shard mapping is updated and errors are thrown when a worker tries to fetch segments from a now stale preprocessor. Workers will then update their shard mapping and retry the fetch from the currently assigned preprocessor.

Figure 10b shows the rate that DAG execution jobs are started and finished in this region. The fluctuations in those rates are typical variations. The close relationship between



(a) Error rate. (b) Throughput and success rate. (c) End-to-end latency.

Figure 10: The effects of injecting a failure of 20% of the preprocessors in a region.



(a) Scheduling error. (b) DAG throughput. (c) DAG completion time.

Figure 11: The effects of a natural failure of 5% of workers within a 1k+ worker pool in a region.

the start and finish rate demonstrates that the success rate and the throughput of SVE are unaffected by the failures. Figure 10c shows the median (P50) and 95th percentile (P95) end-to-end latency (A–E) in this region at this time. This demonstrates that latency in SVE is unaffected by large-scale preprocessor failures.

The second scenario is shown in Figure 11. In this scenario, 5% of workers in a 1000+ machine pool within a region gradually fail. The failures begin at 9:40 with a second spike of failures at 10:30. All workers recovered by 11:45. Figure 11a shows the error rate at schedulers in this region. Schedulers throw errors when they timeout waiting for a worker to report completing an assigned task. The error rate closely tracks the failure of workers with a lag proportional to the max timeout for each job. These timeouts are set per task based on the expected processing time. This explains the gradual decrease in error rate: They are distributed around a mean of a few minutes with a decreasing proportion at longer times.

Figure 11b shows the rate that DAG execution jobs are started and finished in this region. The decrease in start rate is a natural fluctuation not associated with these failures. The close tracking of finish rate to start rate demonstrates the success rate and throughput of SVE are unaffected by this failure of 5% of workers. Figure 11c shows the median (P50) and 95th percentile (P95) end-to-end latency (A–E) in this region at this time. The resilience of SVE to large-scale worker failures is demonstrated by end-to-end latency being unaffected.

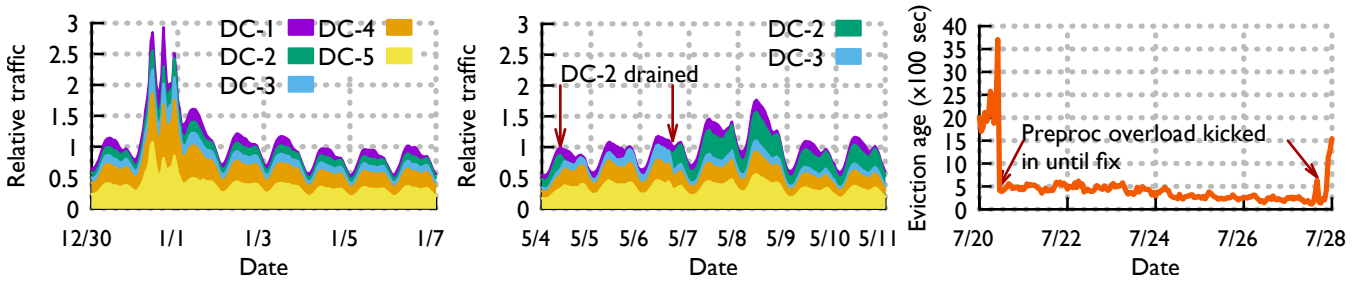
7 OVERLOAD CONTROL

SVE is considered *overloaded* when there is higher demand for processing than the provisioned throughput of the system. Reacting to overload and taming it allows us to continue to reliably process and share videos quickly. This section describes the three sources of overload, explains how we mitigate it with escalating reactions, and reviews production examples of overload control.

7.1 Sources of Overload

There are three primary sources of overload for SVE: organic, load-testing, and bugs. Organic traffic for sharing videos follows a diurnal pattern that is characterized by daily and weekly peaks of activity. Figure 12b shows load from 5/4–5/11. A daily peak is seen each day and the weekly peak is seen on 5/8. SVE is provisioned to handle the weekly peak.

Organic overload occurs when social events result in many people uploading videos at the same time at a rate much higher than the usual weekly peak. For instance, the ice bucket challenge resulted in organic overload. Some organic overload events are predictable, for instance, New Years Eve



(a) Organic overload in all DCs. **(b) Load test overload in DC-3.** **(c) Bug-induced overload.**

Figure 12: The effects of organic, load-test, bug-induced overload events.

2015 saw a 3× increase in uploads over the daily peak. Other organic overload events are not, for instance, the coup attempt in Turkey on July 15, 2016, saw a 1.2× increase in uploads to the European data center as many people in Turkey uploaded videos or went live.

Load-testing overload occurs when our load-testing framework Kraken [32] tests the entire Facebook stack including SVE. The load-testing framework regularly runs a disaster tolerance test that emulates a datacenter going offline, which results in traffic redirection to our other datacenters. This can result in some datacenters (typically the one nearest the drained datacenter) having demand higher than weekly peak load. Our load testing framework helps us know that our overload controls work by testing them regularly. We monitor throughput and latency under this load so we can back it off anytime it negatively affects users.

Bug-induced overload occurs when a bug in SVE results in an overload scenario. One example is when a memory leak in the preprocessor tier caused the memory utilization to max out. This in turn resulted in evictions from the segment cache, which is how we detected the bug. Another example is when an ffmpeg version change suddenly resulted in workers becoming much slower in processing videos due to a change in ffmpeg's default behavior. In this scenario, we saw multiple types of overload control kick in until we were able to roll back the bad push.

7.2 Mitigating Overload

We measure load in each component of SVE and then trigger reactions to help mitigate it when a component becomes overloaded. We monitor load in the CPU-bound workers and memory-bound preprocessors. The front-end and storage tiers, which are used across many applications, separately manage and react to overload. While video processing is computation and memory intense, the DAG complexity is relatively small. Given that our design keeps the scheduler separate from application data, it is unlikely to be overloaded before the workers and preprocessors.

The reactions to overload generally occur in the following order. First, SVE delays latency-insensitive tasks. Each scheduler monitors the CPU load on the workers in its region. When it schedules a task in its queue it checks the current load against the "moderate" threshold. If the current load is over the threshold and the task is marked as latency-insensitive, then the scheduler moves the task to the end of the queue again instead of scheduling it.

If delaying latency-insensitive tasks does not alleviate the overload in a single region, then SVE will redirect a portion of new uploads to other regions. The precise trigger for this behavior is more complex than the logical trigger of worker CPU utilization being high. First, some latency sensitive tasks will be pushed back by the scheduler to avoid thrashing workers. This in turn fires a regional overload alert to the on-call engineer. The on-call engineer then will typically manually redirect a fraction of uploads by updating a map of video traffic to datacenters. In addition, there is an automatic mechanism that will kick in and redirect traffic at a more extreme rate of delaying latency sensitive tasks. Redirecting uploads to other regions allows the overload to be handled by less loaded workers in those other regions. We redirect new uploads, instead of scheduling the tasks in other regions, to co-locate the workers for those videos with the preprocessors that will cache their segments. This co-location reduces latency for those uploads by avoiding cross-region fetching of segments by workers. The co-location also keeps the memory utilization on the preprocessor tier in the overloaded region from maxing out and triggering our final reaction to overload.

Our final reaction to overload is to delay processing newly uploaded videos entirely. If all regions are building queues of latency-sensitive videos then the memory in the preprocessor tier will fill up as it caches segments for videos at a rate greater than they can be processed and evicted from the cache. Videos are spread across preprocessors and each preprocessor detects and reacts to this form of overload. When a preprocessor nears its limit for memory it will start delaying

DAG execution jobs. These jobs are not preprocessed or sent to the scheduler. Instead they are forwarded to storage where they will be kept until the preprocessor is less loaded.

Delaying newly uploaded videos is not ideal because we want our users to be able to share their videos quickly. However, given complete overload of the system it is inevitable that some uploads will be delayed. Delaying whole uploads allows the system to operate at its maximum throughput on the videos it does process. Not delaying whole videos, in contrast, would result in thrashing behavior in the preprocessor cache. This thrashing would decrease the throughput of the system and increase the latency of all videos.

While the reactions to overload are generally triggered in top-to-bottom order, this is not always the case. The trigger condition for each reaction is separately monitored and enforced. For instance, the memory leak bug triggered load shedding through delaying some newly uploaded videos without affecting scheduling.

7.3 Overload Control in Action

Figure 12 shows overload control in action for three different overload scenarios. Figure 12a shows a large spike in organic traffic during New Years Eve 2015. The number of completed DAG execution jobs is shown relative to daily peak (1×). There are three spikes in the graph that correspond roughly to midnight in Australia and Eastern Asia, Europe, and the Americas. The first reaction, delaying latency-insensitive tasks, was triggered for datacenters hitting their max throughput. We anticipated this spike in traffic and so the uploads were already configured to be spread across more datacenters than is typical. This prevented the redirection overload reaction from kicking in by, in essence, anticipating we would need it and proactively executing it. With the load spread across all datacenters and latency insensitive tasks delayed, DAG execution jobs were processed fast enough that the memory on preprocessors did not hit the last threshold and we did not need to delay any newly uploaded videos. The effect of overload control is seen in the figure through the mix of traffic spikes in different datacenters. This demonstrates SVE is capable of handling large organic load spikes that exceed the provisioned capacity of the global system.

Figure 12b shows overload from load testing. Specifically, a disaster readiness test drained all traffic from datacenter 2. This caused the upload traffic for that datacenter to be redirected to other datacenters, with most of the traffic going to datacenter 3, which is the nearest. This demonstrates that SVE is capable of handling load spikes that exceed the provisioned capacity of a given datacenter. In addition, it demonstrates that SVE is resilient to a datacenter failure.

Figure 12c shows overload caused by a memory leak on preprocessors. Once the memory usage exceeded the threshold, preprocessors shed new uploads to storage, and the cache eviction age on the preprocessors dropped from its typically 30+ minutes to a few minutes. The job completion rate (not shown) is unaffected. This demonstrates that SVE is capable of handling large bug-induced load spikes.

8 PRODUCTION LESSONS

This section shares lesson learned from our experience of operating SVE in production for over a year. We share these in the hope they will be helpful to the community in giving more context about design trade-offs and scaling challenges for practical video processing systems at massive scale.

8.1 Mismatch for Livestreaming

Livestreaming is streaming a video from one to many users as it is being recorded. Our livestream video application's requirements are a mismatch for what SVE provides.

The primary mismatch between SVE and livestreaming stems from the overlap of recording the video with all other stages of the full video pipeline in livestreaming. This paces the upload and processing for the video to the rate it is recorded—i.e., each second only 1 second of video needs to be uploaded and only 1 second of video needs to be processed. As a result, upload throughput is not a bottleneck as long as the client's upstream bandwidth can keep pace with the live video. And parallel processing is unnecessary because there is only a small segment of video to process at a given time. Another mismatch is that the flexibility afforded through dynamic generation of a DAG for each video is unnecessary for livestreaming. A third mismatch is that SVE recovers from failures and processes all segments of video, while once a segment is no longer "live" it is unnecessary to process it. Each of these mechanisms in SVE that is unnecessary for livestreaming adds some latency, which is at odds with livestreaming's primary requirement.

As a result, livestreaming at Facebook is handled by a separate system, whose primary design consists of a flat tier of encoders that allocate a dedicated connection and computation slot for a given live stream. Reflecting on this design mismatch helped us realize the similarity between the live encoders and SVE's worker. We are currently exploring consolidating these two components to enable shared capacity management. An important challenge that remains is balancing the load between the two types of workloads: streamed dedicated live encoding and batched full video processing.

8.2 Failures from Global Inconsistency

When an upload begins it is routed from the client to a particular region and then a particular front-end and from

there to a particular preprocessor. To improve locality, this preprocessor will typically handle all uploaded segments of a video. In all cases, once a video is bound to a preprocessor it will be processed entirely in the preprocessor's region. The binding of a video to a preprocessor/region was originally stored in an eventually-consistent geo-replicated data store. For the vast majority of uploads where all segments are routed to the same region this did not cause a problem. When the global load balancing system, however, routed some parts of the upload to a different region the eventually-consistent storage did cause a problem.

If the binding from video to preprocessor/region had not yet replicated to this different region, then the front-end that received a part of the upload did not know where to send it. Once we determined the root cause we coped with it by introducing a retry loop that continues to check for the binding and delays the upload until the binding is replicated.

This short-term solution is not ideal, however, because it leaves the system vulnerable to spikes in replication lag. Many parts of our system—e.g., timeouts on processing requests on front-end machines—are tuned to handle requests in a timely manner. When replication lag spikes, it can delay an upload an arbitrarily long time, which then breaks the expectations in our tuning and can still fail the upload. Thus while our short-term fix handles the problem in the normal case, we have latent failures that will be exposed by slow replication. We are exploring a long-term solution that will avoid this problem by using strongly-consistent storage.

8.3 Failures from Regional Inconsistency

We also used a regional cache-only option of the same replicated store as the part of our intermediate storage (discussed in §3) that is used to pass metadata between different tasks. We picked the cache-only option to avoid needing backing storage, which we thought was a good choice given the metadata is only needed while a DAG job is being executed. Both the consistency of the data store and the cache-only choice have caused us significant maintenance problems.

The data store provides read-after-write consistency within a cluster, which is a subset of a region. Once our pool of workers grew larger than a cluster, we started seeing exceptions as intermediate metadata would either not exist or be stale that would fail the upload. We initially coped with this problem using retries, but this is fragile for reasons similar to those discussed above and did not solve the cache-only problem.

The cache-only option can, and occasionally does, lose data. When this happened it would also cause an upload to fail. We initially coped with this problem by rerunning earlier tasks to regenerate the missing metadata. This was not ideal, however, because rerunning those tasks could take seconds to minutes that would increase end-to-end latency and was a waste of worker resources. We have solved both problems by moving to a persistent database for the intermediate metadata storage. While this increases latency on the order of tens of milliseconds per operation, this small increase in latency is worth it to make our video uploads more robust.

8.4 Continuous Sandboxing

For security we sandbox the execution of tasks within SVE using Linux namespaces. Initially we would create a new sandbox for each task. When we moved our messaging use case onto SVE we saw a considerable spike in latency due to setting up unique network namespaces for each execution. The videos for the messaging use case tend to be smaller and as a result there are typically more of them being concurrently processed on each worker. We found that setting up many sandboxes concurrently caused the spike in latency. We solved this problem by modifying our system to reuse pre-created namespace for sandboxing across tasks, which we call *continuous* sandboxing. We are now investigating more potential efficiency improvements from making more of our system continuous.

9 RELATED WORK

This section reviews three categories of related work: video processing at scale, batch processing systems, and stream processing systems. SVE occupies a unique point in the intersection of these areas because it is a production system that specializes data ingestion, parallel processing, its programming abstraction, fault tolerance, and overload control for videos at massive scale.

Video Processing at Scale. ExCamera [11] is a system for parallel video encoding that achieves very low latency through massively parallel processing of tiny segments of video that can still achieve high compression through state-passing. SVE is a much broader system than ExCamera—e.g., it deals with data ingestion, many different video codecs, programming many video processing applications. SVE could potentially lower its latency further, especially for the highest quality and largest videos, by adopting an approach similar to ExCamera.

A few companies have given high-level descriptions of their video processing systems. Netflix has a series of blog posts [1, 33] that describes their video processing system. Their workload is an interesting counterpoint to ours: they have far fewer videos, with much less stringent latency requirements, and very predictable load. YouTube describes a method to reduce artifacts from stitching together separately processed video segments, which includes a high level

description of their parallel processing [17]. This paper provides a far more detailed description of a production video processing system that this prior work.

Other recent work focus on efficiently using limited computational resources to improve video streaming or querying. Chess-VPS [30] is a popularity prediction service targeted at SVE and Facebook's video workload. Chess-VPS aims to guide the re-encoding of videos that will become popular to enable higher quality streaming for more users. VideoStorm [37] targets a different setting where it makes intelligent decisions on how to process queries over live videos in real time.

Batch Processing Systems. There has been a tremendous boom in batch processing systems and related research in the last fifteen years. MapReduce [10] is a parallel programming model, system design, and implementation that helped kickstart this resurgence. MapReduce's insights were to build the hard parts of distributed computation—fault tolerance, moving data, scheduling—into the framework so application programmers do not need to worry about them and to have the programmer explicitly specify parallelism through their map and reduce tasks. SVE, and many other distributed processing systems, exploit these same insights.

Dryad [14, 34] generalized the map-reduce programming model to DAGs and introduced optimizations such as shared-memory channels between vertices to guarantee they would be scheduled on the same worker. SVE also has a DAG programming model and SVE's task group annotation is equivalent to connecting that set of vertices with shared-memory channels in Dryad. Piccolo [25] is a batch processing system that keeps data in memory to provide lower latency for computations. SVE's caching of segments in preprocessor memory is similar and helps provide low latency.

CIEL [21] is a batch processing system that can execute iterative and recursive computations expressed in the Skywriting scripting language. Spark [35] is a batch processing system that uses resilient distributed datasets to keep data in memory and share it across computations, which results in much lower latency for iterative and interactive processing. Naiad [20] is a batch and stream processing system that introduced the timely dataflow model that allows iterative computations through cycles in its processing DAG. Computations in SVE are neither iterative, recursive, nor interactive.

Stream Processing Systems. There is a significant body of work on stream processing systems that start with early systems like TelegraphCQ [8], STREAM [18], Aurora [3], and Borealis [2]. These systems, in contrast to batch processing systems, consider data ingestion latency implicitly in their model of computation. Their goal is to compute query results across streams of input as quickly as possible. It is thus natural for these systems to consider data ingestion latency.

More recent work [4, 9, 15, 16, 22, 26, 36] extends stream processing in new directions. Spark Streaming [36] focuses on faster failure recovery and straggler mitigation by moving from the continuous query model to modeling stream processing as (typically very small) batch jobs. JetStream [26] looks at wide-area stream processing and exploits aggregation and degradation to reduce scarce wide-area bandwidth. StreamScope [16] is a streaming system at Microsoft with some extra similarities to SVE as compared to typical stream processing systems. StreamScope is designed to make development and debugging of applications easy, as is SVE. Its evaluation of a click-fraud detection DAG evaluates end-to-end latency. This is a rare instance in the stream processing literature where the implicit consideration of data ingestion is made explicit. We suspect these additional similarities arose because both StreamScope and SVE are in production.

10 CONCLUSION

SVE is a parallel processing framework that specializes data ingestion, parallel processing, the programming interface, fault tolerance, and overload control for videos at massive scale. It provides better latency, flexibility, and robustness than the MES it replaced. SVE has been in production since the fall of 2015. While it has demonstrated its resilience to faults and overloads, our experience operating it provides lessons for building even more resilient systems.

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers of the SOSP program committee, our shepherd Geoff Voelker, Zahaib Akhtar, Daniel Suo, Haoyu Zhang, Avery Ching, and Kaushik Veeraraghavan whose extensive comments substantially improved this work. We are also grateful to Eran Ambar, Gux Luxton, Linpeng Tang, Pradeep Sharma, and other colleagues at Facebook who made contributions to the project at different stages.

REFERENCES

[1] Anne Aaron and David Ronca. 2015. High Quality Video Encoding at Scale. http://techblog.netflix.com/2015/12/high-quality-video-encoding-at-scale.html. (2015).

[2] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. 2005. The Design of the Borealis Stream Processing Engine. In *Proceedings of the 2005 Conference on Innovative Data Systems Research*.

[3] Daniel J Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. 2003. Aurora: a new model and architecture for data stream management. *The VLDB Journal–The International Journal on Very Large Data Bases* 12, 2 (2003), 120–139.

[4] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: Fault-Tolerant Stream Processing

at Internet Scale. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1033–1044.

[5] Apache Storm 2017. http://storm.apache.org/. (2017).

[6] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. 2010. Finding a Needle in Haystack: Facebook's Photo Storage. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*. USENIX.

[7] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2013 USENIX Annual Technical Conference*. USENIX.

[8] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J Franklin, Joseph M Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R Madden, Fred Reiss, and Mehul A Shah. 2003. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. ACM.

[9] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M Hellerstein, Khaled Elmeleegy, and Russell Sears. 2010. MapReduce Online. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation*. USENIX.

[10] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*. USENIX.

[11] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation*. USENIX.

[12] HipHop Virtual Machine 2017. http://hhvm.com/. (2017).

[13] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C. Li. 2013. An Analysis of Facebook Photo Caching. In *Proceedings of the 24th ACM Symposium on Operating System Principles*. ACM.

[14] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *Proceedings of the 2nd ACM SIGOPS European Conference on Computer Systems*. ACM.

[15] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthikeyan Ramasamy, and Siddarth Taneja. 2015. Twitter Heron: Stream Processing at Scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM.

[16] Wei Lin, Zhengping Qian, Junwei Xu, Sen Yang, Jingren Zhou, and Lidong Zhou. 2016. StreamScope: Continuous Reliable Distributed Processing of Big Data Streams. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation*. USENIX.

[17] Yao-Chung Lin, Hugh Denman, and Anil Kokaram. 2015. Multipass Encoding for Reducing Pulsing Artifacts in Cloud Based Video Transcoding. In *Proceedings of the 2015 IEEE International Conference on Image Processing*. IEEE.

[18] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. 2003. Query Processing, Resource Management, and Approximation in a Data Stream Management System. In *Proceedings of the 2003 Conference on Innovative Data Systems Research*.

[19] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, et al. 2014. f4: Facebook's Warm BLOB Storage System. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*. USENIX.

[20] Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A Timely Dataflow System. In *Proceedings of the 24th ACM Symposium on Operating System Principles*. ACM.

[21] Derek G Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. 2011. CIEL: a universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation*. USENIX.

[22] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. 2010. S4: Distributed Stream Computing Platform. In *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*. IEEE.

[23] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. 2006. Rethink the Sync. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*. USENIX.

[24] Edmund B Nightingale, Kaushik Veeraraghavan, Peter M Chen, and Jason Flinn. 2008. Rethink the Sync. *ACM Transactions on Computer Systems (TOCS)* 26, 3 (2008), 6.

[25] Russell Power and Jinyang Li. 2010. Piccolo: Building Fast, Distributed Programs with Partitioned Tables. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*. USENIX.

[26] Ariel Rabkin, Matvey Arye, Siddhartha Sen, Vivek S Pai, and Michael J Freedman. 2014. Aggregation and Degradation in JetStream: Streaming Analytics in the Wide Area. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation*. USENIX.

[27] Vijay Rao and Edwin Smith. 2016. Facebook's new front-end server design delivers on performance without sucking up power. https://code.facebook.com/posts/1711485769063510. (2016).

[28] Alyson Shontell. 2015. Facebook is now generating 8 billion video views per day from just 500 million people âĂŤ here's how that's possible. https://tinyurl.com/yc3jhxuu. (2015).

[29] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. 2015. RIPQ: Advanced Photo Caching on Flash for Facebook. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*. USENIX.

[30] Linpeng Tang, Qi Huang, Amit Puntambekar, Ymir Vigfusson, Wyatt Lloyd, and Kai Li. 2017. Popularity Prediction of Facebook Videos for Higher Quality Streaming. In *Proceedings of the 2017 USENIX Annual Technical Conference*. USENIX.

[31] The Hack Programming Language 2017. http://hacklang.org/. (2017).

[32] Kaushik Veeraraghavan, Justin Meza, David Chou, Wonho Kim, Sonia Margulis, Scott Michelson, Rajesh Nishtalaand Daniel Obenshain, Dmitri Perelman, and Yee Jiun Song. 2016. Kraken: Leveraging Live Traffic Tests to Identify and Resolve Resource Utilization Bottlenecks in Large Scale Web Services. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*. USENIX.

[33] Rick Wong, Zhan Chen, Anne Aaron, Megha Manohara, and Darrell Denlinger. 2016. Chelsea: Encoding in the Fast Lane. http://techblog.netflix.com/2016/07/chelsea-encoding-in-fast-lane.html. (2016).

[34] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. 2008. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*. USENIX.

[35] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation*. USENIX.

[36] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *Proceedings of the 24th ACM Symposium on Operating System Principles*. ACM.

[37] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J. Freedman. 2017. Live Video Analytics at Scale with Approximation and Delay-Tolerance.. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation*. USENIX.