VFS: A File System for Video Analytics

ABSTRACT

We present a new video file system (VFS) designed to decouple high-level video operations such as machine learning and computer vision from the low-level details required to store and efficiently retrieve video data. Using VFS, users read and write video data as if it were to an ordinary file system, and VFS: (1) transparently and automatically arranges the data on disk in an efficient, granular format; (2) caches frequently-retrieved regions in the most useful formats; and (3) eliminates redundancies found in videos captured from multiple cameras with overlapping fields of view. Our results suggest that this approach can improve read performance by up to 54%, reduce storage costs by up to 45%, and enable developers to focus on application logic rather than video storage and retrieval.

1 INTRODUCTION

Driven by advances in computer vision and machine learning, along with the proliferation of cameras in the world around us, the volume of video data captured and processed is rapidly increasing. YouTube receives more than 400 hours of uploaded video per minute [39], and more than six million closed-circuit television cameras populate the United Kingdom, collectively amassing an estimated 7.5 petabytes of video data per day [6]. Law-enforcement use of body-worn cameras is expected to exceed 200,000 units in service by the end of 2019 [18], collectively generating almost a terabyte of video data per day [42]. A single autonomous vehicle can generate more than 19 terabytes of video data per *hour* [14].

To support this video data deluge, many systems and applications have emerged to ingest, transform, and reason about such data [12, 16, 20, 22, 23, 27, 33, 43]. Each of these systems, however, stores data on disk as large, opaque, and independent blobs. As a result, storage-related operations on video data are inflexible and limited to reads, writes, and coarse-grained seeks. These limited storage capabilities create three sets of challenges for application developers.

First, video-oriented applications are forced to tightly intermingle data plumbing with application logic. Developers must manually handle the (de)compression of physically-persisted video data, resolution resampling before applying machine learning and computer vision algorithms, and frame rate subsampling in network-constrained and edge-processing applications. This intermixing leads to applications that are brittle and difficult to evolve.

Second, the close coupling of application logic with physical video data storage makes it difficult to deploy optimizations and other techniques to improve application performance. For example, applications that select or produce multiple versions of a video (e.g., at different resolutions or by selecting multiple regions of interest) are responsible for (re)compressing, persisting, and selecting from amongst the potentially many versions when performing subsequent operations. Many applications avoid this additional complexity by inefficiently reapplying operations to the original video data.

Finally, many recent applications collect large amounts of video data with overlapping fields of view and physically proximate locations. For example, traffic camera and surveillance networks often have multiple cameras oriented toward the same intersection. Similarly, autonomous driving and drone applications come with overlapping multiple sensors that capture nearby video data. Reducing the redundancies that occur among these sets of physically-proximate or otherwise similar video streams is neglected in all modern video-oriented systems. Leveraging spatial overlap between videos could improve application performance, but doing so is difficult given that developers need to devote considerable effort dealing with low-level video compression idiosyncrasies.

In this paper, we introduce a new *video file system (VFS)*, which is designed to decouple application design from video data's physical layout and compression optimizations. This decoupling allows application and system developers to focus on their relevant functionality, while VFS handles the low-level details associated with video data persistence.

Analogous to relational database management systems, developers using VFS treat each video as a *logical video*, and let VFS determine the best way to perform operations over one or more of the *physical videos* that it maintains on disk. To enable this independence, VFS exposes a simple and familiar file-system interface where users read and write video data through POSIX file system operations or by using VFS's command line or C++ API. Users initially write video data in any format, encoding, and resolution—either compressed or uncompressed—and VFS manages the underlying compression, serialization, and physical layout on disk. When users subsequently read video—once again in any configuration and by optionally specifying regions of interest and other selection criteria—VFS automatically identifies and leverages the most efficient methods to retrieve and return the requested data.

```
# Write initial video
$ vfs write traffic.mp4
# Read small RGB version
$ vfs read traffic.rgb -r 320x180
# Read 1K RGB between time 5 and 20
$ vfs read traffic.rgb -r 1280x720 -s 5 -e 20
# Read 1K RGB cropped at 640x720 between time 5 and 20
$ vfs read traffic.rgb -r 1280x720 -s 5 -e 20 -w 640 -h 720
# Read the same video as above using the POSIX interface
$ cat /vfs/traffic/1280x720s5e20x640y720.rgb > traffic.rgb
```

Figure 1: Example commands using the VFS command-line interface to read and write traffic video data.

The VFS interface frees users from worrying about video formats, compression, sampling, and other low-level persistence details. A user writes data in whatever video format is on hand, and later reads video data in whatever format is needed, *even if she has not previously written video in that format to VFS.*

Under the hood, VFS deploys the following optimizations and caching mechanisms to improve read and write performance. First, rather than storing video data on disk as a single, opaque blob, VFS decomposes a video into sequences of contiguous, independently-decodable, optionally-compressed sets of frames. VFS then indexes those video fragments and the resulting structure allow VFS to read only the minimal set of subsequences necessary to satisfy a request for portions of a video. As VFS handles requests for video over time, it maintains a per-video cache that is populated passively as a byproduct of read operations. When a user performs a read operation, VFS leverages a minimal-cost subset of these views to generate its answer. Because video fragments can arbitrarily overlap, VFS uses a SMT solver to identify the best sets of views to satisfy a request.

Second, to trade off the size of the cache with its associated storage costs, VFS exposes an application-specified video storage budget. This budget allows administrators to balance between these factors. When a video's storage budget is exceeded, VFS prunes stale cache entries by selecting those least likely to be useful in answering subsequent queries and, among equivalent entries, VFS optimizes for quality and defragmentation.

Finally, VFS reduces the storage cost of redundant video data collected from physically proximate cameras. It does so by deploying a *joint compression* optimization that identifies overlapping regions of video and stores the overlapping region only once. Developers may explicitly indicate videos eligible for joint compression, or may allow VFS to automatically identify eligible video data.

Collectively, by decoupling video-oriented systems and applications from the underlying physical representation of video data, VFS allows for easier application development, faster read and write performance for computer vision and machine learning pipelines, and lower storage costs for physically proximate video datasets. In summary, we make the following contributions:

- We design a new file system for video data that leverages the physical properties of videos to improve application performance (Section 2).
- We describe a novel technique to perform reads by selecting from potentially many cache entries to efficiently produce an output while maintaining the quality of the resulting video data (Section 3).
- We describe ways to optimize the storage required to persist videos that are highly overlapping or contain similar visual information (Section 5) and a protocol for caching multiple versions of the same video (Section 4).
- We evaluate VFS against existing video storage techniques and demonstrate that it can reduce the time of video reads by up to 54% and decrease storage requirements by up to 45% (Section 6).

2 VFS OVERVIEW

Consider an application designed to monitor an intersection for license plates associated with missing children or adults with dementia. A typical implementation of such an application would ingest video data from multiple locations around the intersection, decompress it, and convert it to an alternate representation suitable for input to a machine learning model trained to detect license plates. Such models are typically deep learning models that are expensive to execute. Therefore, the application might first use an inexpensive low-resolution video representation to prune frames that are unlikely to contain license plates at all. VStore [41], for example, implements this type of optimization, but requires the application specify beforehand that it needs both a low- and high-resolution approximations. Other image processing systems use similar optimizations [22, 28]. Once vehicles with potentially matching plates are found in the video, the application might apply another machine learning algorithm to cross reference the vehicle make and model. This third processing step may require the video in yet another representation. Finally, a user might request and view all video sequences containing likely candidates. This might involve further converting to a representation compatible with the viewer (e.g., at a resolution compatible with a mobile device or compressed using a supported codec).

An application working with video files stored in a regular file system must manually convert to and from these various representations, while an application that utilizes a system like VStore must decide a priori every possible format or

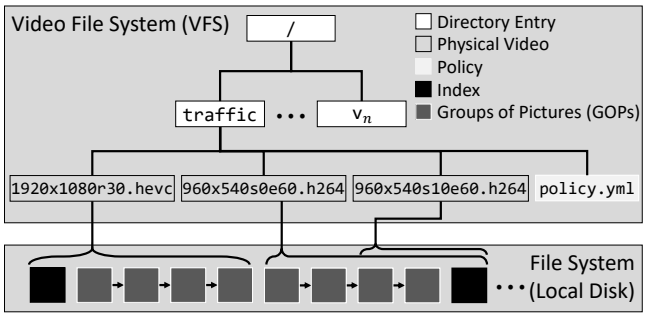


Figure 2: The video file system (VFS) directory structure. Each video is organized under its own directory entry, which contains physical videos and a policy file. Each physical video is associated with a sequence of optionally-compressed video segments called groups of pictures (GOPs). Adapted from [Anonymized].

resolution that might be needed in the future. On the other hand, an application leveraging VFS can simply read video data in the desired form. For example, when the above application wants low-resolution, uncompressed video data to find frames with license plates, it can directly read the file `traffic/320x180.rgb` from VFS (or execute the command line variant `vfs read traffic.rgb -r 320x180`), letting VFS be responsible for efficiently producing the desired data.

Figure 1 shows additional examples of VFS commands using the VFS command-line interface. A `vfs write` command ingests video data into VFS, which creates a logical entry for the video (if it doesn't already exist) and writes the video data as an initial physical video. The `vfs read` command produces a new file on disk containing the requested video fragment at the desired resolution, times, and physical format (if it has not already been cached).

Critically, VFS automatically selects the most efficient way to generate the desired video data in the requested format based on the original video and cached representations. For example, only certain regions of frames generally contain license plates. If the low-resolution heuristic identifies such a region, the application can explicitly request just that region, e.g., `traffic/1280x720x640y360.rgb`, requests a 1280×720 resolution video cropped to the region defined by the rectangle at (0, 0) and (640, 360).

Table 1 summarizes the full set of operations that VFS exposes. Importantly, these operations are over *logical videos*, which VFS executes to produce or store *physical video* data. Each operation involves a point- or range-based scan or insertion over a single logical video source. VFS allows constraints on any combination of temporal (T), spatial (S), and physical (P) parameters. Temporal parameters include start and end time interval ($[s, e]$) and frame rate (f); spatial parameters include resolution ($r_x \times r_y$) and region of interest ($[x_0..x_1]$ and $[y_0..y_1]$); and physical parameters P include physical frame layout (l; e.g., YUV420, YUV422) and compression method (c; e.g., HEVC).

Table 1: VFS operations. Both reads and writes require specification of spatial (S; resolution, region of interest), temporal (T; start and end time, frame rate), and physical (P; frame layout, compression codec) parameters.

Op	Parameters
read	$(name, S, T, P)$
write	$(name, S, T, P, data)$
create	$(name)$
delete	$(name)$
policy	$(name, parameter, value)$
pin	$(name, S, T, P)$
unpin	$(name, S, T, P)$

Besides the POSIX-compliant file system interface (Table 1), VFS also provide a C++, Python, and OpenCV-compatible API, and a command-line interface similar to that exposed by HDFS [36] (Figure 1). Our prototype treats the POSIX interface as primary, and each of the other APIs delegate to it.

An exemplary directory hierarchy of VFS is illustrated in Figure 2. To create the missing-person application described above, a user simply executes `vfs write` or writes each traffic camera's video using the POSIX interface (e.g., to `/vfs/traffic.mp4`). In both cases, VFS implicitly executes *create*(`traffic`), which creates a new logical video and directory, `traffic`, on the file system. It then automatically extracts the video's metadata for spatial, temporal, and physical parameters, and uses these to execute *write*(`traffic`, S, T, P). For example, if the video `traffic.mp4` contained video data at 2K resolution, 30 frames per second, and YUV420 format, the `vfs write` would create a new file `traffic/1920x1080r30fyuv420.hevc`. In cases where VFS cannot automatically extract video parameters (e.g., when writing raw uncompressed data), the user must specify them explicitly.

Later, the application may read this or other representations of the logical video from VFS. Critically, rather than being able to read only the previously-written variant, users may read the logical video using *any* combination of valid parameters and are not limited to reading previously-written results (e.g., opening and reading `/traffic/999x999r99.h264` would constitute a valid read). Regardless of what an application reads, VFS transparently manages decompression and resampling, freeing the application from needing to be involved.

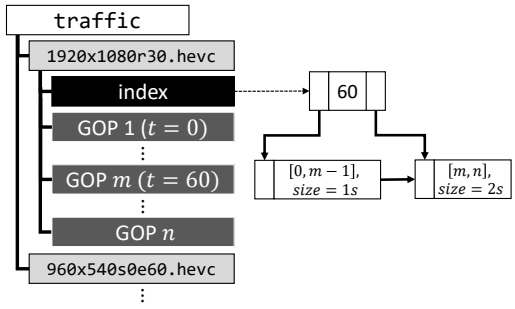


Figure 3: An example VFS physical organization that contains one logical video and two underlying physical videos. For physical video `1920x1080r30.hevc`, the first m GOPs are each one second in length, while the remaining $n - m$ are two seconds. These durations are recorded in the associated index. Shading is as shown in Figure 2.

Under the hood, VFS arranges each physical video as a sequence of entities called *groups of pictures (GOPs)*. Each GOP is composed of a contiguous sequence of frames in the same format and resolution. A GOP may contain raw pixel data or be compressed using a video codec. Compressed GOPs, however, are constrained such that they are independently decodable and may not take data dependencies on other GOPs.

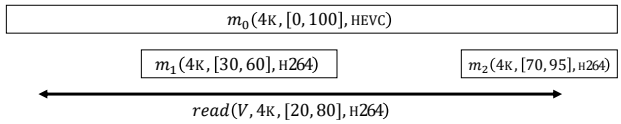
Though a GOP may contain an unbounded number of frames, video compression codecs typically fix their size to a small, constant number of frames (30–300) and VFS accepts as-is ingested compressed GOP sizes (which are typically less than 512kB). However, partitioning extremely long GOPs on ingest is a straightforward enhancement. For uncompressed GOPs, our prototype implementation automatically partitions video data into blocks of size $\leq$ 25MB (the size of a single RGB 4K frame), or a single frame for resolutions that exceed this threshold.

Figure 3 illustrates the internal physical state of VFS after executing the write described above. VFS has created a directory that contains the previously-written physical representation for the logical video `traffic`. VFS has stored each GOP in this representation on disk as a series of distinct files `v/1920x1080r30.hevc/1`, ..., `v/1920x1080r30.hevc/n`. It has also constructed a non-clustered temporal index that maps from time to the GOP file containing visual information for that time. This level of detail is invisible to applications, which access VFS only through the operations summarized in Table 1 and also illustrated in Figure 1.

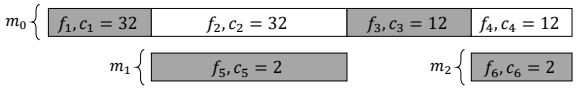
Finally, each logical video is also associated with a *policy* that determines the hyperparameters (listed in Table 2) used to tune read and write performance. Policies are modified using the *policy* API method shown in Table 1. Developers may also *pin* (and *unpin*) regions of a logical video to prevent their being evicted from VFS's internal cache. Pinned regions are not counted against the video's storage budget.

Table 2: Video policy parameters and default values.

Parameter	Default	Description
Quality threshold	35	Maximum quality degradation (PSNR)
Storage budget	10×	Per-video cache size
Joint compression	$\emptyset$	Joint compression candidates



(a) Read operation on three cached physical videos



(b) Physical video fragments with simplified cost formulae

Figure 4: Figure 4(a) shows the query $read(V, 4K, [20, 80], H264)$ executed over a video with cached physical videos $\{m_0, m_1, m_2\}$. Figure 4(b) shows the weighted physical video fragments using simplified cost formulae. The lowest-cost result is shaded.

3 DATA RETRIEVAL FROM VFS

As mentioned, VFS internally represents each logical video as a collection of one or more cached physical videos. When executing a read, VFS must produce the result using one or more of these physical videos.

Consider a simplified version of the alert application described in Section 2, where a single camera has captured 100 minutes of HEVC-encoded video and written it to VFS using the name V. The application first reads the entire video and applies a computer vision algorithm that identifies two regions (at minutes 30–60 and 70–95) containing license plates. The application then retrieves those fragments again requesting H264 compression to transmit to a device that only supports this format. As a result of these operations, VFS now contains the original video (m_0) and the cached versions of the two fragments (m_1, m_2) as illustrated in Figure 4(a). The figure indicates the labels $\{m_0, m_1, m_2\}$ of the three videos, their spatial configuration (4K), start and end times (e.g., $[0, 100]$ for m_0), and physical format (HEVC or H264).

Later, a first responder on the scene views a one-hour portion of the recorded video on her mobile device, which only has hardware support for H264 decompression. To deliver this video, the application

executes *read*(V, 4K, [20, 80], H264), which, as shown at the bottom of Figure 4(a), requests video V between time [20, 80] in spatial configuration 4K and physical configuration H264.

VFS responds by first identifying subsets of the available physical videos that can be leveraged to produce the result. For example, VFS can simply transcode m_0 between times [20, 80]. Alternatively, it can transcode m_0 between time [20, 30] and [60, 70], m_1 between [30, 60], and m_2 between [70, 80]. The latter plan is the most efficient since m_1 and m_2 are already in the desired output format (H264), hence VFS need not incur a heavy transcoding cost for these regions. Figure 4(b) shows the different selections that VFS might make to answer this read. Each *physical video fragment* $\{f_1, ..f_6\}$ in Figure 4(b) represents a different region that VFS might select. Note that VFS need not consider other subdivisions of these fragments—for example by subdividing f_5 at time [30, 40] and [40, 60]—since f_5 being cheaper at [30, 40] implies that it is at [40, 60] too.

To model these transcoding costs, VFS employs a *transcode cost model* $c_t(f)$ that represents the cost of converting a physical video fragment f from a source physical format into a target physical format.

VFS must also ensure that the quality of a result has sufficient fidelity. For example, using a heavily downsampled physical video (e.g., 32×32 pixels) to answer a read requesting 4K video is likely to be unsatisfactory. To avoid this, VFS adopts a quality model $u(f_0, f)$ that gives the expected quality loss of using a fragment f in a read operation relative to using the originally-written video f_0. When considering using a fragment f in answering a read, VFS will reject it if the expected quality loss is below a user-specified cutoff: $u(f_0, f) < \epsilon$. The user optionally specifies this cutoff in the read's physical parameters (see Table 1); otherwise a default threshold is used ($\epsilon = 35$ in our prototype). The range of u is a non-negative peak signal-to-noise ratio (PSNR), a common measure of quality variation based on mean-squared error [15]. Values ≥40 are considered to be lossless qualities, while ≥30 are near-lossless.

Given this cost and quality model, we now turn to how VFS selects fragments for use in performing a read operation. In general, given a *read* operation and a set of physical videos, producing a result requires VFS to perform several operations. First, it must select fragments that cover the desired spatial and temporal ranges. To ensure that a solution exists, VFS maintains the initially-written video (the *root physical video* m_0), and VFS returns an error for reads extending outside of the temporal interval of m_0.

Second, when the selected physical videos temporally overlap, VFS must resolve which physical video fragments to use in producing the answer in a way that minimizes the total conversion cost of the selected set of video fragments. This problem is reminiscent of prior work in materialized view selection [11]. Fortunately, a VFS read is far simpler than a general database query, and in particular is constrained to a small number of parameters with point- or range-based predicates.

We motivate our initial solution by continuing our example from Figure 4(a). First, observe that the collective start and end points of the physical videos form a set of *transition points* where VFS may opt to switch to an alternate physical video.

In Figure 4(a), the transition times include those in the set {30, 60, 70}, and we illustrate them in Figure 4(b) by partitioning the set of cached physical videos at each transition point. We also omit fragments that are outside the read's temporal range, since they do not provide information relevant to the read operation.

Now observe that between each consecutive pair of transition points VFS must choose exactly one physical video fragment. In Figure 4(b), we highlight one such set of choices that covers the read interval. Each choice of a fragment comes with a cost (i.e., f_1 has cost 32), derived using a cost formula that assigns a transcode cost equal to (i) twice the fragment's duration if it requires decoding and then re-encoding, (ii) the fragment's duration if it only requires encoding, and (iii) zero for fragments already in the target physical format P_q (e.g., H264 for the example in Figure 4). Formally:

$$c_t(f) = \begin{cases} 0 & \text{if } \textsc{physical-format}(f) = P_q \\ \textsc{duration}(f) & \text{if } \textsc{uncompressed}(f) \\ 2 \cdot \textsc{duration}(f) & \text{otherwise} \end{cases}$$

To conclude our example, observe that we must choose a set of physical video fragments that (i) cover the queried temporal range, and (ii) do not temporally overlap. Further, of all the possible paths between 20 and 80, the one with the lowest cost—highlighted in Figure 4(b)—minimizes the total cost of producing the answer. These characteristics collectively meet the requirements identified at the beginning of this section.

3.1 Interval Cover Fragment Selection

We propose two algorithms to solve the cost minimization problem. Both find optimal fragment selections, with the first targeting videos without *dependent frames* (to be discussed in Section 3.2).

Our first algorithm models the problem as a weighted interval cover problem, with each fragment being temporally arranged on a real-number line. Weights for each fragment are given by the cost function c, and only fragments admitted by the quality model u are considered. Given a VFS read, we identify the minimum-cost set of physical view fragments as follows:

Problem (Cached physical video fragment interval cover)**.** Consider a $read(D, S_q, T_q, P_q)$ operation over logical video D, which is associated with cached physical videos $M = \{m_1, ..., m_n\}$ each having configuration (S_i, T_i, P_i). We are given a function $C(m, T)$ that converts physical video m into configuration (S_q, T, P_q) at cost $c(m, T)$, and a function $\oplus$ that concatenates physical videos with identical configurations. We want to find a result $A = C(m_0^*, T_0^*) \oplus \cdots \oplus C(m_k^*, T_k^*)$ that is:

- **Valid** such that it is *non-overlapping* (i.e., $T_i^* \cap T_j^* = \emptyset$) and is a *cover* of T (i.e., $\bigcup T_i^* = T_q$).
- **Minimal cost** such that $\forall\{(m, T) \mid m \in M, T \subseteq T_q\}: \sum c(m_i, T_i) \geq c(A)$.

Definitions. Let (s_i, e_i) be the start and end time of physical video m_i and (s_q, e_q) be the start and end times of the read operation (i.e., T_q). Let $W = \{s_q, e_q, s_1, e_1, ..., s_n, e_n\}$ be the set of all such start and end times. Finally, let $R_i = \{p \mid p \in W \times W, \max(s_i, s_q) \leq p \leq \min(e_i, e_q)\}$ be the transition points associated with m_i.

Solution. We begin by decomposing each physical video into fragments defined on the intervals $F_i = \{[\alpha, \beta] \mid \alpha, \beta \in R_i, \alpha < \beta, \nexists\gamma \in R_i : \alpha < \gamma < \beta\}$. The size $|F_i|$ is at most $O(|M|)$, the number of physical videos. Collectively, let $F = \bigcup F_i$ be the set of all fragments, and $|F| = O(|M|^2)$. We associate weight $c(f)$ to each fragment $f \in F$.

Using the fragments, which cover the interval (s_q, e_q), we wish to find a minimum-weight cover. While covering problems are NP-hard in most contexts, interval covering problems are solvable in polynomial time [34]. As illustrated in Figure 4, our formulation of fragment selection involves selecting minimum-cost intervals on the (temporal) positive real number line. Since no intervals partially overlap, we sort the intervals by start time and sequentially select the fragment with lowest cost. This approach is a special case of the greedy weighted interval cover algorithm, which generates a minimum-weight cover R on the interval (s_q, e_q) [2]. Applying this algorithm has complexity linear in the number of intervals, or $O(|F|) = O(|M|^2)$.

3.2 Constraint Satisfaction Selection

The previous solution is inexpensive and optimal for selecting subsets of videos that have transcode cost proportional to duration or number of frames. However, as video compression codecs utilize data dependencies between frames, this assumption is frequently violated. Consider the illustration in Figure 5, which shows the frames within a physical video with their data dependencies indicated by directed edges. If VFS wishes to begin a fragment at the frame highlighted in red, it must first decode all of the red frame's *dependent frames*, as shown in gray. This violates

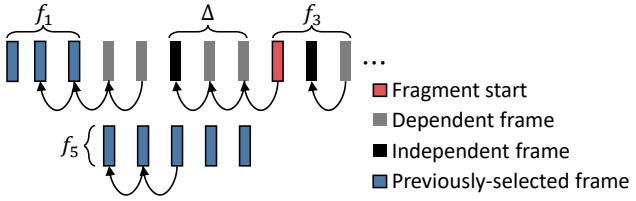


Figure 5: In this simplified illustration based on the fragments $\{f_1, f_3, f_5\}$ in Figure 4, VFS is considering using fragment f_3 starting with the frame highlighted in red, and has already decided to use f_1 and f_5. However, this frame cannot be decoded without also transitively decoding the dependencies represented by directed edges and labeled Δ. VFS look-back cost c_l is a function of these frames, with the single independent frame being more expensive than the two dependent frames.

the assumption in the transcode cost model above: the cost of transcoding frame depends on where within the video it occurs, and whether its dependent frames are also transcoded.

To model this cost, we add a *look-back cost* $c_l(P, f)$ that gives the cost of decoding the frames in fragment f along with the transitive frames dependencies $\Delta = \{\delta_1, ..., \delta_n\}$ not part of f and *if they have not already been decoded*, meaning that they are not a member of the previously-selected fragments P. As illustrated in Figure 5, these dependencies come in two forms: independent frames $\mathrm{A} \subseteq \Delta$ (i.e., frames with out-degree zero) which are more expensive to transcode, and dependent frames (those with outgoing edges) which are inexpensive but require first decoding dependencies. Our prototype weighs independent frames to be $\eta = 5$ times more expensive than dependent frames. Formally, we set $c_l(P, f) = \eta \cdot |\mathrm{A} - P| + |\Delta - \mathrm{A} - P|$.

Optimizing the VFS *robust cost model* $c_r(f) = c_f(P, f) + c_t(f)$ requires jointly optimizing both c_f and c_t, where each fragment choice affects the transitive dependencies P of future choices. This problem is not solvable in polynomial time, so we employed a SMT solver [7] in order to generate an optimal solution to c_r. Our embedding constrains frames in overlapping fragments so that only one is chosen, and uses information about the locations of independent and dependent frames in each physical video to compute the cumulative decoding cost due to both transcode and look-back for any set of selected fragments. Solving this is typically more expensive than the interval cover problem, and we evaluate our two algorithms in Section 6.1.

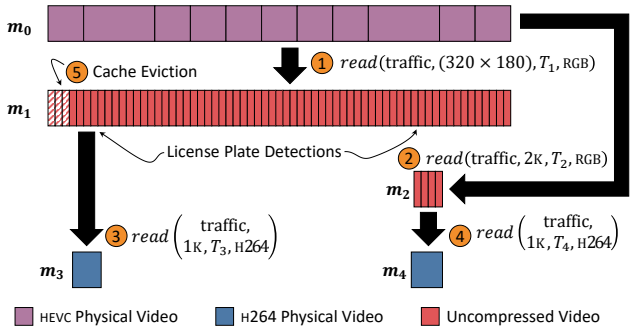


Figure 6: VFS caches read results and uses them to answer future queries. In ❶ an application reads logical video `traffic` at 320×180 resolution for use in license plate detection (see command two in Figure 1) and VFS caches the result as m_1. In ❷ VFS caches m_2, a region with a dubious license plate detection (command 3 in Figure 1). In ❸ and ❹ VFS caches H264-encoded m_3 & m_4, where license plates were detected. However, reading m_4 exceeds the storage budget and VFS responds by evicting the striped region at ❺.

4 DATA CACHING IN VFS

In the previous sections we described how VFS persists, reads, and writes physical videos. We now describe how VFS decides *which* physical videos to maintain, and which to evict under low disk space conditions. The caching process involves making two interrelated decisions:

- When executing a read operation, should VFS admit the result as a new physical video for use in answering future reads?
- When disk space grows scarce, which existing physical video(s) should VFS discard?

To aid in both of these decisions, VFS maintains a video-specific *storage budget* that limits the total size of the physical videos associated with each logical video. The storage budget is stored in each video's policy (see Table 2) and may be specified as a multiple of the size of the initially-written physical video or a fixed ceiling in bytes. This value is initially set to an administrator-specified default (10× the size of the root physical video in our prototype). VFS never evicts the root physical video (see Section 3), which is always available in the cache. Pinned regions of a logical video (see Section 2) are not counted against a logical video's storage budget, and VFS does not consider pinned regions for cache eviction.

As a running example, consider the sequence of reads illustrated in Figure 6, which mirrors the alert application commands shown in Figure 1 and described in Section 2. In this example, an application reads a low-resolution uncompressed video from VFS for use with a license detection algorithm. VFS caches the result as a sequence of three-frame GOPs (approximately 518kB per GOP). One detection was marginal, and so the application reads higher-quality 2K video to apply a more accurate detection model. VFS caches this result as a sequence of single-frame GOPs, since each 2K RGB frame is 6MB in size. Finally, the application extracts two H264-encoded regions for offline viewing. VFS caches m_3, but when executing the last read it determines that it has exceeded its storage budget and must now decide whether to cache m_4.

The key idea behind VFS's cache is to logically break physical videos into "pages." That is, rather than treating each physical video as a monolithic cache entry, VFS targets the individual GOPs *within* each physical video. Using GOPs as cache pages greatly homogenizes the sizes of the entries that VFS must consider. VFS's ability to evict GOP pages *within* a physical video differs from other variable-sized caching efforts such as those used by content delivery networks (CDNs), which are forced to make decisions on large, indivisible, and opaque entries (a far more challenging problem space with limited progress in formulating approximate solutions [5]).

However, there are several key differences between GOPs and pages. In particular, GOPs are related to each other:

- One GOP might be a higher-resolution version of another.
- Consecutive GOPs form a contiguous video fragment.

These correlations make typical eviction policies like least-recently used (LRU) problematic. In particular, application of naïve LRU might evict every other GOP in a physical video, decomposing it into many small fragments and increasing the cost of reads (which have quadratic complexity in the number of fragments; see Section 3).

Additionally, given multiple, redundant GOPs that are all variations of one another, ordinary LRU would treat eviction of a redundant GOP the same as any other GOP. However, our intuition is that it is desirable to treat redundant GOPs different than singleton GOPs without such redundancy.

Given this intuition, our eviction policy modifies LRU in the following ways:

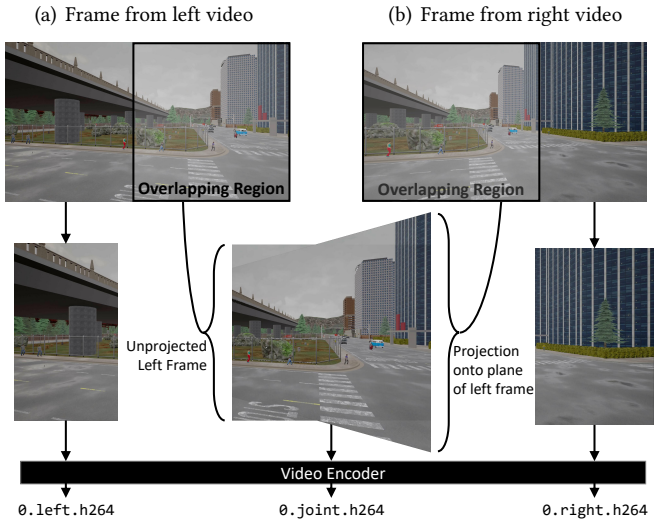
- We decrease the recency of GOPs occurring at the beginning and end of a physical video more highly than those in the middle.
- We decrease the recency of the lowest-quality (as given by our quality cost model; see Section 3) GOP that has higher-quality redundant copies.

In Figure 6, we show VFS choosing to evict the three-frame GOP at the beginning of m_1 and to cache m_4. If our prototype had weighed the second modification more heavily than the first, VFS would instead elect to evict m_3, since it was not recently used and is the variant with lowest quality.

5 DATA COMPRESSION IN VFS

As described in Section 2, when an application writes data to VFS, VFS partitions the written video into blocks by GOP (for compressed video data) or contiguous frames (for uncompressed video data). VFS follows the same process when caching the result of a read operation for future use.

To improve the storage performance of written and cached video data, VFS employs two compression-oriented optimizations and one optimization that reduces the number of physical video fragments, each of which we describe in this section. Specifically, VFS (i) jointly compresses redundant data across multiple physical videos (Section 5.1); (ii) lazily performs compression on blocks of uncompressed, infrequently-accessed GOPs (Section 5.2); and (iii) improves the performance of reads by compacting temporally-adjacent physical videos (Section 5.3).



(c) Left, overlapped, and right regions are separately encoded.

Figure 7: Joint compression applied to two horizontally-overlapping frames. VFS first identifies the overlapping regions in each frame, combines them, and encodes the three resulting pieces (left, overlap, and right) separately.

5.1 Joint Physical Video Compression

Many video applications capture video from cameras that are spatially proximate with similar orientations. For example, a bank of traffic cameras mounted on a pole with each capture video of the same intersection from similar angles. Despite the redundant information that mutually exists in these video streams, most applications treat these video streams as distinct and persist them separately to disk.

VFS optimizes storage of these videos by reducing the redundancy between pairs of highly-similar video streams. This optimization, which we call *joint compression*, is applied between pairs of *logical videos* written to VFS. In applying the optimization, VFS matches the cached physical videos associated with each logical video. For each pair of frames in a matching logical video, it identifies a region of overlap between the frames and combines them. This process is transparent to users, who can continue to logically read the individual videos as if they were stored separately on disk.

Figure 7 illustrates this process on two frames taken from a synthetic dataset (Visual Road-1K-50%, described in Section 6). Figure 7(a) and Figure 7(b) respectively show the two frames with the overlapping regions highlighted, and Figure 7(c) shows the overlapping regions combined.

Critically, because the frames were captured from cameras at different orientations, combining them requires more than an isomorphic translation or rotation (e.g., the angle of the horizontal sidewalk is not aligned in the two frames). Instead, a homography between the two frames is estimated and the result used to transform between the two spaces. As shown in Figure 7(c), VFS applies this transformation to the right frame which causes its right side to bulge vertically. However, after it is overlaid onto the left frame the two align near-perfectly.

More specifically, VFS applies joint projection by executing the following steps. First, it estimates a homography between the two frames. To do so, it waits to receive two frames from the pair of videos being jointly compressed (or reads two frames from video pairs being lazily compressed). Next, it applies a feature detection algorithm (SIFT [26]) that identifies similar features that co-occur in both frames. Using these features and random sample consensus (RANSAC [9]), it estimates the homography matrix used to transform between frame spaces. Our current prototype applies SIFT and RANSAC every thirty frames.

With the homography estimated, VFS uses it to transform the right frame into the space of the left frame. This results in three distinct regions: (i) a "left" region of the left frame that does not overlap with the right, (ii) an overlapping region, and (iii) a "right" region of the right frame that does not overlap with the left. VFS splits these into three distinct regions and uses an ordinary video codec to encode each region separately.

This process is formalized in Algorithm 1. The function JOINT-COMPRESS iterates over each frame in videos F and G and estimates homography every thirty frames. It then uses the JOINT-COMPRESS-FRAMES to perform pairwise joint compression on each frame f and g by computing x_f, the right extent of the overlap in the left frame and x_g, the left

Algorithm 1 Joint compression algorithm

```
let HOMOGRAPHY(f, g) estimate the 3×3 homography matrix of f and g
let PARTITION(f, [x_0, x_1]) be the subframe defined by x_0 ≤ x < x_1
let MERGE(f, g) overlay frame g onto f

function JOINT-COMPRESS(F, G)
    Input: Video frames F = {f_1, ..., f_n}
    Input: Video frames G = {g_1, ..., g_n}
    Output: Vector of compressed (left, merged, right) tuples
    C ← ∅
    for i ∈ [1..n] do
        if i mod 30 = 0 then
            H ← HOMOGRAPHY(f, g)
            if H_{1,2} < 0 then F, G ← G, F
        C ← C ⊕ JOINT-COMPRESS-FRAMES(f_i, g_i, H, i)
    return C

function JOINT-COMPRESS-FRAMES(f, g, H)
    x_f ← [H^{-1} · (0  0  WIDTH(f))]_{0,2}
    x_g ← [H · (0  0  1)]_{0,2}
    l ← PARTITION(f, [0, x_f))          ▷ left part of f
    m_f ← PARTITION(f, [x_f, WIDTH(f)))  ▷ overlap of f
    m_g ← PARTITION(g, [0, x_g))         ▷ overlap of g
    r ← PARTITION(g, [x_g, WIDTH(f_g)))  ▷ right part of g
    m ← MERGE(m_f, H · m_g)
    return (COMPRESS(l), COMPRESS(m), COMPRESS(r))
```

extent of the overlap in the right frame. It uses these cut-points to subdivide f and g into left, overlapping, and right regions and compresses each individually.

This joint compression optimization process is activated in one of two ways. First, a user may explicitly request joint compression between two videos by modifying a pair of videos' policies (see Table 2). Even when specified in the policy, VFS will only apply joint compression when the estimated homography indicates that the amount of overlap exceeds an administrator-specified threshold (25% in our prototype).

Alternatively, VFS lazily examines pairs of physical videos and applies joint compression by selecting frames of candidate physical videos and evaluating them for overlap. To prevent false positives, when lazily compressing physical videos VFS applies joint compression only for pairs of physical videos that have substantial overlap with high RANSAC confidence, as specified as an administrator-configurable threshold.

5.2 Deferred Compression

Most video-oriented applications operate over decoded video data (e.g., RGB). Such data is vastly larger than its compressed counterpart: storage for a typical 4K video exceeds five terabytes per hour when uncompressed (e.g,. the VisualRoad-4K-30% dataset we describe in Section 6 is 5.2TB uncompressed as 8-bit RGB). As VFS caches uncompressed video regions as the result of reads (e.g., while scanning for license plates), it may quickly exhaust the video's storage budget.

One option is to perform lossless compression on uncompressed frames prior to caching them. However, doing so when executing the read increases the operation latency and leads to decreased performance for applications that request only small amounts of high-resolution uncompressed frames. An alternative approach would be to begin performing lossless compression during reads when the associated video's storage budget exceeds some threshold. For example, VFS might allocate half of its budget for uncompressed data and compress cached entries after this amount is exceeded. This would increase read latency in exchange for improved cache performance.

This approach, however, suffers from two disadvantages. First, it never prompts the system to compress the data that was cached prior to the budget threshold being exceeded. Second, the new read may produce a cache entry that is less useful for future reads than an alternative cache entry (e.g., it might be immediately evicted). This means that the computational expense of compression is wasted.

To avoid these disadvantages, VFS adopts the following approach. When a video's cache size exceeds some threshold (25% in our prototype), VFS activates a special *deferred compression* mode. In this mode, when a read requests uncompressed data, VFS examines the current cache and orders the uncompressed physical video entries by eviction order. It then performs lossless compression on the *last* entry in this list (i.e., the entry least likely to be evicted). It then executes the read as usual.

Our prototype uses Zstandard for lossless compression, which emphasizes compression and decompression speed but has a lower compression ratio relative to more expensive image and video codecs such as PNG and HEVC [8].

VFS performs two additional optimizations beyond the approach described above. First, Zstandard comes with a "compression level" setting, which is an integer in the range [1..19], with the lowest setting having the fastest speed but the lowest compression ratio (and the highest setting having the opposite characteristics). VFS linearly scales Zstandard compression level with remaining storage budget, which has the effect of decreasing compressed size while increasing compression time. Second, while deferred compression is active, VFS continues to compress cache entries in a background thread during periods when no other IO requests are being executed.

Algorithm 2 Contiguous materialized view compaction

Ensure: All pairs of contiguous physical videos compacted
  let HARDLINK(f, d) create hard link for file f in dir d
  let INSERT(m, f) insert file f into the temporal index of m
  let DIR(m) be the directory associated with m
  let VIEWS(v) be the physical videos associated with v
  let CONFIG$_{s,p}(m)$ give the spatial/physical config of m
  let s_{m_i}, e_{m_i} be the start and end times of m_i

  for all $v_1 \in VFS, v_2 \in VFS \setminus v_1$ **do**
    for all $m_1 \in$ VIEWS$(v_1), m_2 \in$ VIEWS(v_2) **do**
      if $e_{m_1}{=}s_{m_2} \wedge$CONFIG$_{s,p}(m_1) =$ CONFIG$_{s,p}(m_2)$ **then**
        let $F_2 =$ DIR(m_2) sorted by time
        for all $f_2 \in F_2$ **do**
          HARDLINK$(f_2,$ DIR$(m_1))$
          INSERT(m_1, f_2)
        DELETE(m_2)

5.3 Physical Video Compaction

As a result of caching the result of user queries, VFS may persist pairs of cached videos that contain data in contiguous time and with the same spatial and physical configurations. For example, the cached reads resulting over a logical video at time [0, 90] and [90, 120] are contiguous. Application of lazy compression may also create contiguous physical videos. For example, if VFS lazily compressed an uncompressed physical video starting at time 120, it would be contiguous with the cached video covering time [90, 120].

To reduce the number of physical videos that need to be considered in performing a read, VFS periodically compacts pairs of contiguous cached videos and substitutes a unified representation. To do so non-quiescently, it applies the algorithm shown in Algorithm 2. This algorithm examines pairs of cached videos and, for each contiguous pair, uses hard links to merge the GOPs from the second into the first. The result is a unified cached video that contains the aggregated video data from both sources.

6 EVALUATION

We have implemented a prototype of VFS using approximately 5,000 lines of C++ code. GPU-based operators were implemented using CUDA [32] and NVENCODE/NVDECODE [31]. We use the `libfuse` file system in userspace (FUSE) reference implementation to export VFS to the operating system kernel and expose the POSIX-compliant interface for consuming applications [25]. VFS also uses FFmpeg [4] for some video plumbing operations such as GOP segmentation and concatenation. Our prototype currently adopts a no-overwrite policy for logical videos and disallows updates. We plan on supporting both of these features in a future release. Finally, when performing writes, VFS does not guarantee that data are visible to other readers until the file being written is closed.

Table 3: Datasets used to evaluate VFS

Dataset	Resolution	# Frames	Compressed Size (MB)
Robotcar	1280×960	7,494	120
Waymo	1920×1280	398	7
VisualRoad 1K-30%	960×540	108k	224
VisualRoad 1K-50%	960×540	108k	232
VisualRoad 1K-75%	960×540	108k	226
VisualRoad 2K-30%	1920×1080	108k	818
VisualRoad 4K-30%	3840×2160	108k	5,500

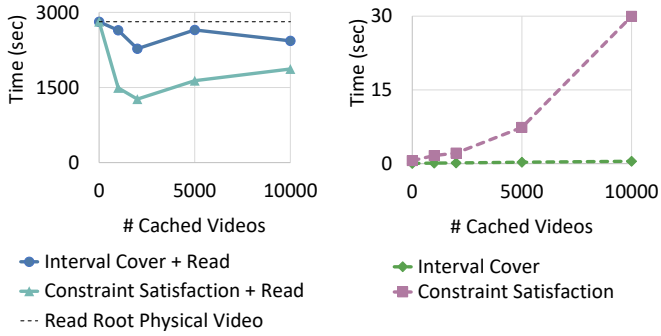
We evaluate VFS by comparing it to two baseline systems (and directly against the local file system) in terms of read (Section 6.1), write and caching (Section 6.2), and compression (Section 6.3) performance.

Baseline systems. We compare VFS against VStore [41], a recent storage system that supports video analytic workloads by pre-computing all possible video representations. We also evaluate VFS against direct use of the local file system.

We build VStore with support for GPU-accelerated video encoding and decoding, and where available utilize these accelerated operations. We experienced intermittent failures when running VStore on >2,000 frame videos, and to work around this all experiments on VStore are limited to this size. The local file system system is formatted using `ext4` and backed by a SSD drive.

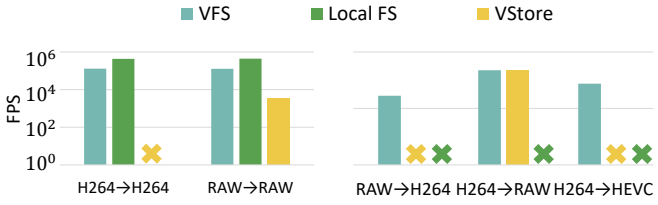
Experimental configuration. We perform all experiments using a single-node system equipped with an Intel i7-6800K processor with 6 cores running at 3.4Ghz and 32GB DDR4 RAM. The system also includes a Nvidia P5000 GPU with two discrete NVENCODE chipsets.

Datasets. In our evaluation, we use a combination of real and synthetic video data. We use the former to measure VFS performance under real-world inputs, while the latter allows us to test on a variety of carefully-controlled configurations. We use the datasets shown in Table 3 for the experiments throughout this section. The "Robotcar" dataset consists of two highly-overlapping videos captured using adjacent stereo cameras mounted on a moving vehicle [29]. The dataset is provided as 7,494 separate images, which we converted into a video using H264 at 30 frames per second and one-second GOPs.



(a) Time to select fragments and read (b) Time to select video fragments

Figure 8: Total time to select fragments and perform a read with varying number of physical videos.



(a) Read in same format (b) Read in different format

Figure 9: Read throughput in frames per second for the Visual Road 1K-30% dataset for VFS and baseline systems. Each group $I \rightarrow O$ shows throughput reading video in format I and outputting it in format O. An × means that a system does not support the read type (e.g., the local file system does not support reading raw video as H264).

The "Waymo" dataset is an autonomous driving dataset [38]. We selected one segment (approximately twenty seconds) from the dataset, which was captured using two vehicle-mounted cameras. Unlike the Robotcar dataset, we estimate that Waymo videos overlap by approximately 15%.

Finally, the various "VisualRoad" datasets consist of synthetic video generated using a recent video analytics benchmark designed to evaluate the performance of video-oriented data management systems [13]. To generate each dataset, we used Visual Road to generate a one-hour simulation and produce video data at at 1K, 2K, and 4K resolutions. We also modified the field of view of each panoramic camera in the simulation so that we could vary the horizontal overlap of the resulting videos. We repeated this process several times and produced five distinct datasets; for example, the "VisualRoad-1K-75%" dataset contains two one-hour videos, where each video has 75% horizontal overlap with the other.

Because the size of the uncompressed 4K Visual Road datasets exceed the storage capacity of our experimental system, we do not show results for this dataset that require fully persisting its uncompressed representation to disk.

6.1 Data Retrieval Performance

Read Performance. Our first experiment explores the read performance of VFS using various numbers of physical videos generated by cached reads. In this experiment, we vary the number and types of fragments available in the cache. First, repeatedly execute queries of the form *read*(VisualRoad-4K-30%, 3840×2160, $[t_1, t_2]$, P), with times $0 \leq t_1 < t_1 + 1 \leq t_2 < 3600$ (in seconds) along with a physical format $P \in \{$H264, HEVC, RGB, YUV420, YUV422, NV12$\}$, with t_1, t_2, and P drawn uniformly at random. These cache entries might be generated by application of a machine learning algorithm (e.g., license plate detection) over many regions of a video. We iterate this process until VFS has cached a variable number of physical videos. For this experiment we assume an infinite storage budget.

We then execute a maximal read (i.e., from time 0 to 3600 seconds) of this dataset in the HEVC format, which is different from the format of the originally-written physical video (H264) and allows VFS to leverage its cached physical video fragments.

We show the performance of this read in Figure 8 for the two application scenarios. Since none of the other baseline systems support automatic conversion from H264 to HEVC, we do not show their runtimes for this experiment.

As we see in Figure 8(a), even a cache with a small number of entries is able to improve read performance by a substantial amount—28% at 100 entries and up to a maximum improvement of 54%. We further observe that the constraint satisfaction algorithm outperforms its interval cover counterpart. This is because the solution it finds requires decoding fewer redundant dependent frames. However, while for 4K video the cost of applying both algorithms does not constitute a significant fraction of the read operation, as we show in Figure 8(b) the exponential cost of constraint satisfaction diminishes the performance benefit as the number of fragments grows large. Switching to interval cover for extremely large caches would likely improve performance in these edge cases, though we leave this as future work.

Read Format Flexibility. Our next experiment evaluates VFS's ability to transparently read video data in a variety of formats. To evaluate this functionality relative to the baseline systems, we first write the VisualRoad-1K-30% dataset to VFS, VStore, and the local disk. We write the data to each file system in both compressed (224MB) and uncompressed form (approximately 328GB).

We use an empty cache for VFS, and read the persisted videos from each system in various formats and measure the throughput offered by each system. Figure 9 shows results for a read in the same format (Figure 9(a)) written to a file system and different formats (Figure 9(b)). Because the local file system and does not support automatic representation transformation (e.g., converting H264-compressed video into RGB), we do not show results for these cases. Additionally, VStore does not support reading some formats from its store, and we additionally omit its result for this case.

We find that read performance *without* a format conversion from VFS is modestly slower than the local file system, due in part to recently-identified bottlenecks in FUSE [37], the local file system being able to execute entirely without kernel transitions, and the need for VFS to concatenate many individual GOPs. However, our results show that VFS is able to adapt to reads in *any* format, a benefit not available when using the local file system.

We additionally find that VFS performance outperforms VStore when reading uncompressed video and is similar when transcoding H264. At the same time, VFS offers more flexible input and output format options and does not require a workload be specified in advance.

Fragment Selection Performance. Our final read experiment evaluates the overhead associated with the interval cover and constraint satisfaction fragment selection algorithms using 30-frame GOPs (i.e., one independent frame and 29 dependent frames). The dashed lines in Figure 8(b) shows the time required to execute each of the algorithms in isolation, without the accompanying read.

We find that the overhead of selecting physical video fragments is low relative to the cost of producing the output, and that the advantage offered by utilizing the physical video fragments outweighs this cost except for a cache with many single-frame entries.

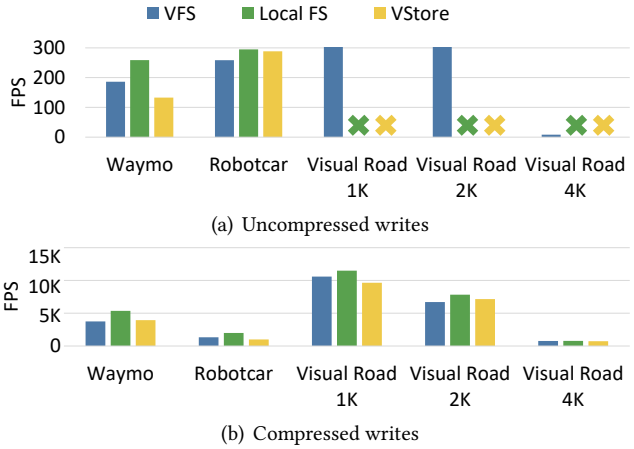


Figure 10: Throughput in frames per second to write uncompressed RGB (Figure 10(a)) and compressed H264 (Figure 10(b)) data to VFS and other baseline systems. An × means that a system does not support the write type (e.g., the local disk lacks capacity to store the >5 terabytes uncompressed VisualRoad-4K-30% dataset).

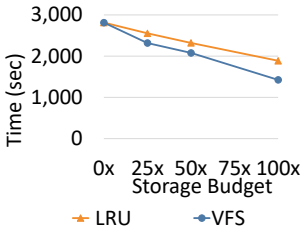


Figure 11: Read runtime by storage budget size for LRU and VFS cache eviction policies.

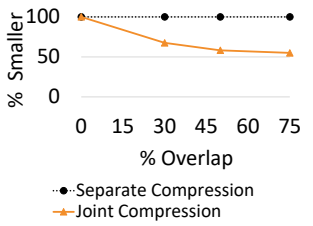


Figure 12: Size of joint compression relative to separately-compressed data.

6.2 Data Persistence & Caching

Write Throughput. Our next evaluation explores VFS write and caching performance. To evaluate the write performance of VFS relative to the other baseline systems, we write each dataset to the respective systems in both compressed and uncompressed form. We measure the write throughput of each system and report the results in Figure 10(a).

For datasets that will fit on local storage, VFS performs similarly to using the local file system and VStore, though VFS outperforms VStore for the extremely small Waymo dataset. On the other hand, none of the baseline systems have the capacity to store the larger uncompressed datasets. For example, the uncompressed VisualRoad-4K-30% dataset is over five terabytes. However, as the video's storage budget reaches capacity, VFS is able to activate its deferred decompression optimization and automatically begin compressing the data being written. This compression allows it to store datasets that no other system can handle, albeit at the cost of decreased throughput.

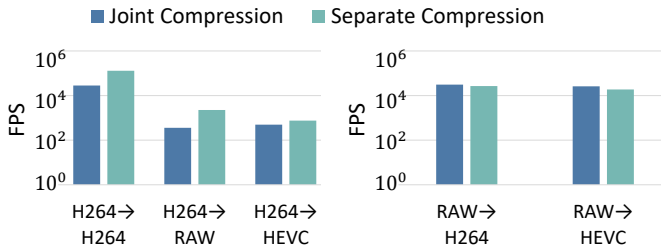
We next write the compressed evaluation datasets to each store. Figure 10(b) shows the performance results for each baseline system. Here all systems perform approximately equivalently, with both VFS and VStore exhibiting minor overhead relative to the local file system.

Cache Performance. To evaluate the VFS cache eviction policy, we repeat our experimental setup for read performance in Section 6.1. We execute 5,000 random read operations to populate the VFS cache. However, instead of assuming an infinite storage budget, we limit it to be various multiples of the input size (e.g., 25×) and apply either the

Table 4: Joint compression recovered quality

Dataset	Quality (PSNR)	
	Left Frame	Right Frame
Robotcar	62**	17
Waymo	32*	29
VisualRoad-1K-30%	40**	30*
VisualRoad-1K-50%	36*	28
VisualRoad-1K-75%	36*	24
VisualRoad-2K-30%	36*	30*
VisualRoad-4K-30%	36*	30*

** lossless quality * near-lossless quality



(a) Read throughput (b) Write throughput

Figure 13: Throughput for writes with and without the joint compression optimization.

least-recently used (LRU) or VFS eviction policy. This has the effect of limiting the number of physical videos available for performing reads.

With the cache populated, we execute a final maximal read for the entire video range (i.e., [0, 3600]). Figure 11 shows the runtimes for each policy and storage budget. These results show that (i) that VFS is able to reduce read execution by approximately 14%, even with a limited budget, relative to application of a LRU policy.

6.3 Compression Performance

Joint Compression Quality. In this evaluation we examine the recovered quality of jointly-compressed physical videos. For this experiment we write various overlapping Visual Road datasets to VFS. We then subsequently read each video back from VFS and compare the resulting quality against its originally-written counterpart. We use the peak signal-to-noise ratio (PSNR) as a quality comparison metric.

Table 4 gives the PSNR for recovered data compared against the written videos. Recall that a PSNR of ≥40 is considered to be lossless, and ≥30 near-lossless [15]. In general, we find high quality recovery for the left input to jointly compressed videos, and near-lossless degradation for the right input. This difference is due to our approach of copying the left frame onto the right and loss in fidelity when performing the inverse projection on the right frame. A potential enhancement, which we leave as future work, is to instead write the mean of the overlapping pixels or to interlace them.

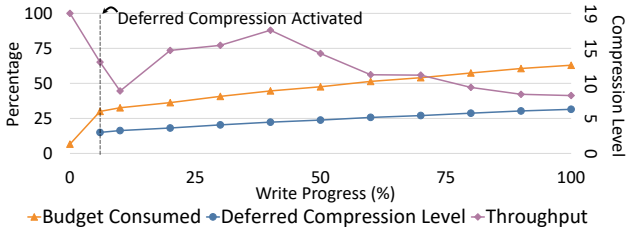


Figure 14: Deferred compression performance when writing the VisualRoad-1K-30% dataset to VFS. At 25% the compression optimization activates and changes the slope of the budget consumed.

Joint Compression Throughput. Our next experiment examines the VFS read throughput with and without the joint compression optimization applied. For this experiment, we write each video in the VisualRoad-1K-30% dataset to VFS, once with joint compression enabled and separately with it disabled. We then execute a read operation in various physical configurations and for the entire duration. Figure 13(a) shows the throughput achieved when executing the read using each configuration. Our results indicate that read overhead for videos stored using joint compression is modest but similar to reads that are not co-compressed.

Applying joint compression to a pair of videos requires a number of nontrivial operations, and our final experiment evaluates the overhead associated with its execution. For this experiment, we write each video in each Visual Road dataset to VFS and measure throughput. Figure 13(b) shows the results of this experiment. Surprisingly, joint writes are *faster* than writing each video stream separately. This speedup is due to VFS's encoding of each of the three lower-resolution streams in parallel, and since compression time is roughly proportional to resolution, encoding the three lower-resolution components is faster than the original frame. Additionally, the overhead in identifying homography (approximately 0.5 milliseconds for every GOP) and partitioning frames (approximately 2 milliseconds applied per-frame) does not obviate this performance advantage.

Deferred Compression Performance. Our write experiment evaluates the performance of deferred compression of uncompressed video writes. To evaluate, we write 3600 frames of the VisualRoad-1K-30% dataset to VFS, leaving storage budget (10× the input, or 2,240MB) and deferred compression configuration (25% threshold) at their defaults. At regular intervals we extract the storage used, Zstandard compression level, and write throughput.

The results for these metrics are listed in Figure 14. We show storage used as a percentage of the budget. Similarly, we show throughput relative to writing without deferred compression activated. Finally, we show compression level

as a value in the interval [1..19]. As expected, storage used exceeds the deferred compression threshold early in the write, and a slope change shows that deferred compression is having a moderating influence on write size. Compression level scales linearly with storage used. Finally, throughput drops substantially as compression is activated, recovers considerably, and then slowly degrades as the compression level is increased.

7 RELATED WORK

Increased interest in applied machine learning and computer vision has lead to the development of a number of new systems that target video analytics, including Optasia [27], LightDB [12], Chameleon [20], and Scanner [33]. These systems all read and write to a local or distributed file system and can immediately leverage the performance advantages afforded by VFS. Video-oriented deep learning accelerators such as BlazeIt [22], VideoStorm [43], Focus [16], and NoScope [23] are similarly complimentary and can transparently benefit from using VFS.

While the systems community has a long history of introducing specialized file systems (e.g., HDFS [36]), few prior systems have targeted video analytics (although other authors have identified the need for systems like VFS [10, 21]). VStore [41] is one such example that targets machine learning workloads by staging video in a pre-specified set of formats. However, VStore requires that the developer to know beforehand the specific workload being optimized, lacks the ability to efficiently read and write beyond these workload formats, and does not take advantage of data independence to improve performance beyond persisting these fixed materializations. Other authors have looked at optimizing on-disk layout of video data in the context of scalable streaming [24]. Finally, related storage-oriented systems such as Haystack [3] and VDMS [35] emphasize image-based operations and metadata access.

Interest in edge processing and networked cameras in the context of video analytics is also emerging, prompting applications that exploit clusters of networked cameras (e.g., VideoEdge [1, 17, 19, 40]). Since these cameras have constrained storage and compute resources, they would benefit from a storage system such as VFS that can transparently balance these factors and improve performance.

Finally, the database community has a long history of exploiting data independence in order to improve performance, which is a key technique used by VFS to obtain its performance advantages. For example, it transparently employs Zstandard [8] compression rather than a typical video codec. Other orthogonal optimizations could be employed to further improve VFS's performance such as Vignette [30] or the homomorphic operators described in LightDB [12].

8 CONCLUSION

In this paper we presented a video file system (VFS) designed to improve the performance of video-oriented applications and data management systems. VFS decouples high-level operations such as computer vision and machine learning algorithms from the low-level plumbing required to read and write data in a suitable format. Users leverage VFS by reading and writing video data in whatever format is most useful, and VFS transparently identifies the most efficient method to retrieve that video data. To maximize interoperability, VFS behaves as if it were an ordinary file system.

We compared our VFS prototype against a recent video storage system and using a standard local file system. Our experiments showed that, relative to both local storage and other dedicated video storage systems, VFS offers more flexible read and write formats and reduces read time by up to 54%. Our optimizations also decrease the cost of persisting video by up to 45%.

REFERENCES

[1] Ganesh Ananthanarayanan, Victor Bahl, Landon P. Cox, Alex Crown, Shadi Nogbahi, and Yuanchao Shu. 2019. Video Analytics - Killer App for Edge Computing. In *MobiSys*. 695–696.

[2] Mikhail J. Atallah, Danny Z. Chen, and DT Lee. 1995. An optimal algorithm for shortest paths on weighted interval and circular-arc graphs, with applications. *Algorithmica* 14, 5 (1995), 429–441.

[3] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. 2010. Finding a Needle in Haystack: Facebook's Photo Storage. In *OSDI*. 47–60.

[4] Fabrice Bellard. 2018. FFmpeg. https://ffmpeg.org.

[5] Daniel S. Berger, Nathan Beckmann, and Mor Harchol-Balter. 2018. Practical Bounds on Optimal Caching with Variable Object Sizes. *POMACS* 2, 2 (2018), 32:1–32:38.

[6] Cloudview. 2018. Visual IoT: Where the IoT Cloud and Big Data Come Together. http://www.cloudview.co/downloads/10. (2018).

[7] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS*. 337–340.

[8] Facebook. [n. d.]. Zstandard real-time compression algorithm. https://facebook.github.io/zstd.

[9] David A. Forsyth and Jean Ponce. 2012. *Computer Vision - A Modern Approach, Second Edition*. Pitman.

[10] Vishakha Gupta-Cledat, Luis Remis, and Christina R. Strong. 2017. Addressing the Dark Side of Vision Research: Storage. In *HotStorage*.

[11] Alon Y. Halevy. 2001. Answering queries using views: A survey. *VLDB* 10, 4 (2001), 270–294.

[12] Brandon Haynes, Amrita Mazumdar, Armin Alaghi, Magdalena Balazinska, Luis Ceze, and Alvin Cheung. 2018. LightDB: A DBMS for Virtual Reality Video. *PVLDB* 11, 10 (2018), 1192–1205.

[13] Brandon Haynes, Amrita Mazumdar, Magdalena Balazinska, Luis Ceze, and Alvin Cheung. 2019. Visual Road: A Video Data Management Benchmark. In *SIGMOD*. 972–987.

[14] Stephan Heinrich and Lucid Motors. 2017. Flash memory in the emerging age of autonomy. *Flash Memory Summit* (2017).

[15] Alain Horé and Djemel Ziou. 2010. Image Quality Metrics: PSNR vs. SSIM. In *ICPR*. 2366–2369.

[16] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodík, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B. Gibbons, and Onur Mutlu. 2018. Focus: Querying Large Video Datasets with Low Latency and Low Cost. In *OSDI*. 269–286.

[17] Chien-Chun Hung, Ganesh Ananthanarayanan, Peter Bodík, Leana Golubchik, Minlan Yu, Paramvir Bahl, and Matthai Philipose. 2018. VideoEdge: Processing Camera Streams using Hierarchical Clusters. In *SEC*. 115–131.

[18] Shelley S Hyland. 2018. Body-worn cameras in law enforcement agencies, 2016. *Bureau of Justice Statistics Publication No. NCJ251775* (2018).

[19] Samvit Jain, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, and Joseph Gonzalez. 2019. Scaling Video Analytics Systems to Large Camera Deployments. In *HotMobile*. 9–14.

[20] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodík, Siddhartha Sen, and Ion Stoica. 2018. Chameleon: scalable adaptation of video analytics. In *SIGCOMM*. 253–266.

[21] Junchen Jiang, Yuhao Zhou, Ganesh Ananthanarayanan, Yuanchao Shu, and Andrew A. Chien. 2019. Networked Cameras Are the New Big Data Clusters *(HotEdgeVideo'19)*. 1–7.

[22] Daniel Kang, Peter Bailis, and Matei Zaharia. 2018. BlazeIt: Fast Exploratory Video Queries using Neural Networks. *CoRR* abs/1805.01046 (2018).

[23] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2017. NoScope: Optimizing Deep CNN-Based Queries over Video Streams at Scale. *PVLDB* 10, 11 (2017), 1586–1597.

[24] Sooyong Kang, Sungwoo Hong, and Youjip Won. 2009. Storage technique for real-time streaming of layered video. *Multimedia Systems* 15, 2 (2009), 63–81.

[25] libfuse [n. d.]. The reference implementation of the Linux FUSE (Filesystem in Userspace) interface. https://github.com/libfuse/libfuse.

[26] David G. Lowe. 1999. Object Recognition from Local Scale-Invariant Features. In *ICCV*. 1150–1157.

[27] Yao Lu, Aakanksha Chowdhery, and Srikanth Kandula. 2016. Optasia: A Relational Platform for Efficient Large-Scale Video Analytics. In *SoCC*. 57–70.

[28] Yao Lu, Aakanksha Chowdhery, Srikanth Kandula, and Surajit Chaudhuri. 2018. Accelerating Machine Learning Inference with Probabilistic Predicates. In *SIGMOD*. 1493–1508.

[29] Will Maddern, Geoff Pascoe, Chris Linegar, and Paul Newman. 2017. 1 Year, 1000km: The Oxford RobotCar Dataset. *IJRR* 36, 1 (2017), 3–15.

[30] Amrita Mazumdar, Brandon Haynes, Magdalena Balazinska, Luis Ceze, Alvin Cheung, and Mark Oskin. 2019. Vignette: Perceptual Compression for Video Storage and Processing Systems. *CoRR* abs/1902.01372 (2019).

[31] nvenc [n. d.]. Nvidia Video codec. https://developer.nvidia.com/nvidia-video-codec-sdk.

[32] NVIDIA Corporation. 2007. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. NVIDIA Corporation.

[33] Alex Poms, Will Crichton, Pat Hanrahan, and Kayvon Fatahalian. 2018. Scanner: efficient video analysis at scale. *TOG* 37, 4 (2018), 138:1–138:13.

[34] Datta Krupa R., Aniket Basu Roy, Minati De, and Sathish Govindarajan. 2017. Demand Hitting and Covering of Intervals. In *ACALDAM*. 267–280.

[35] Luis Remis, Vishakha Gupta-Cledat, Christina R. Strong, and Ragaad Altarawneh. 2018. VDMS: An Efficient Big-Visual-Data Access for Machine Learning Workloads. *CoRR* abs/1810.11832 (2018).

[36] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *MSST*. 1–10.

[37] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. 2017. To FUSE or Not to FUSE: Performance of User-Space File Systems. In *USENIX*. 59–72.

[38] waymo [n. d.]. Waymo Open Dataset. https://waymo.com/open.

[39] Susan Wojcicki. 2018. The Potential Unintended Consequences of Article 13. https://youtube-creators.googleblog.com/2018/11/i-support-goals-of-article-13-i-also.html.

[40] Mengwei Xu, Tiantu Xu, Yunxin Liu, Xuanzhe Liu, Gang Huang, and Felix Xiaozhu Lin. 2019. Supporting Video Queries on Zero-Streaming Cameras. *CoRR* abs/1904.12342 (2019).

[41] Tiantu Xu, Luis Materon Botelho, and Felix Xiaozhu Lin. 2019. VStore: A Data Store for Analytics on Large Videos. In *EuroSys*. 16:1–16:17.

[42] Billy Yates. 2018. Body Worn Cameras: Making Them Mandatory. (2018).

[43] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodík, Matthai Philipose, Paramvir Bahl, and Michael J. Freedman. 2017. Live Video Analytics at Scale with Approximation and Delay-Tolerance. In *NSDI*. 377–392.