TileClipper: Lightweight Selection of Regions of Interest from Videos for Traffic Surveillance

Shubham Chaudhary*, Aryan Taneja*, Anjali Singh+, Purbasha Roy*, Sohum Sikdar*, Mukulika Maity*, Arani Bhattacharya*

*{shubhamch, aryan19027, purbashar, sohum20339, mukulika, arani}@iiitd.ac.in; IIIT-Delhi
+anjali038btcse19@igdtuw.ac.in; Indira Gandhi Delhi Technology University for Women

Abstract

With traffic surveillance increasingly used, thousands of cameras on roads send video feeds to cloud servers to run computer vision algorithms, requiring high bandwidth. State-of-the-art techniques reduce the bandwidth requirement by either sending a limited number of frames/pixels/regions or relying on re-encoding the important parts of the video. This imposes significant overhead on both the camera side and server side compute as re-encoding is expensive. In this work, we propose TILECLIPPER, a system that utilizes tile sampling, where a limited number of rectangular areas within the frames, known as tiles, are sent to the server. TILECLIPPER selects the tiles adaptively by utilizing its correlation with the tile bitrates. We evaluate TILECLIPPER on different datasets having 55 videos in total to show that, on average, our technique reduces $\approx 22\%$ of data sent to the cloud while providing a detection accuracy of 92% with minimal calibration and compute compared to prior works. We show real-time tile filtering of TILECLIPPER even on cheap edge devices like Raspberry Pi 4 and nVidia Jetson Nano. We further create a live deployment of TILECLIPPER to show that it provides over 87% detection accuracy and over 55% bandwidth savings.

1 Introduction

In recent years, real-time traffic surveillance has become important for automatic enforcement of traffic rules [30], control of traffic lights [14], and the detection of anomalous events like accidents [49]. Cities like Shanghai, New Delhi, and New York have installed hundreds of thousands of surveillance cameras[1].The video feeds generated from these cameras are either processed locally or sent to the cloud/edge servers for applying computer vision algorithms. These algorithms run deep neural networks (DNNs), which are inherently compute-intensive. Processing it locally requires expensive hardware (like GPUs and NPUs), thus increasing the cost of traffic surveillance and impacting the scalability. On the other hand, a major challenge faced by techniques that send video feeds to servers is that the amount of data generated is very high, going up to 1Mbps per camera [17], leading to high network bandwidth consumption. Thus, it is essential to find techniques to reduce bandwidth consumption without sacrificing the quality of traffic surveillance.

Current techniques of reducing bandwidth typically utilize one or more of two strategies. The first technique is intelligently selecting the objects or frames of interest that should be sent to the cloud server [17, 38, 61]. However, video is usually encoded such that pixels are defined as an offset of their neighboring pixel values as a compression strategy, where the neighbors could be either spatial or temporal. Thus, only sending the objects or frames of interest would require re-encoding, which mandates either the integration of the algorithm in the camera's firmware [42] or a re-encoding on the device directly connected to the camera [38, 52]. Adding such capability to the cameras or devices attached to it would need a substantial amount of investment. Further, while this technique can save a lot of bandwidth during off-peak hours, it is difficult to save bandwidth when the traffic is congested. The second technique is to perform additional computation on the cloud server by either running more powerful models [54] or sending a signal to the camera to send additional data only when needed [17]. This technique saves bandwidth at the cost of additional GPU usage, which is also expensive and energy-intensive. Thus, a solution that runs *without adding to the computation while also being simple to integrate with existing systems* is essential to reduce the cost of traffic surveillance.

In this paper, we focus on traffic surveillance of *moving* objects such as vehicles, pedestrians, and so on. For this, we avoid the problem of re-encoding proposed in prior works as follows. Recent video standards like HEVC (High-Efficiency Video Codec) or H.265 [50] allow the videos to be split into independently encoded spatial rectangular blocks called tiles (Figure 1b). This allows us to send tiles containing only the objects of interest (moving vehicles or pedestrians) while

[1] https://www.comparitech.com/vpn-privacy/the-worlds-most-surveilled-cities/

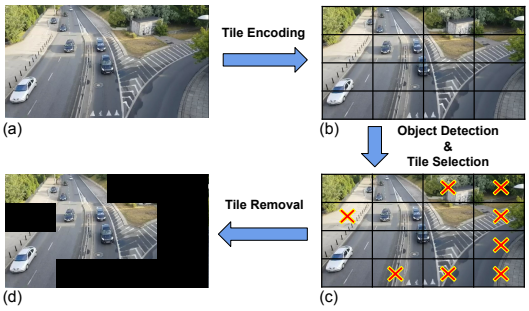


Figure 1: Illustration of TILECLIPPER. Steps: (a) encoding the video in the form of tiles supported by the HEVC format, (b) selecting only the relevant tiles by using the statistical pattern of bitrates, (c) clipping out the unnecessary tiles, and (d) sending this filtered video to the server for surveillance.

omitting the other tiles without re-encoding. Figure 1 shows the workflow of our system. The key advantage of removing tiles is that it runs in real-time, even on embedded platforms like Jetson Nano, unlike techniques that filter frames or objects of interest and re-encode videos. As surveillance cameras with native HEVC support increasingly become available [5], removing tiles is easy to integrate into actually deployed surveillance systems.

However, identifying tiles with objects of interest generally requires running DNNs, which are themselves compute-intensive in nature. The camera usually does not have sufficient compute capability or even power to run any such algorithm. Thus, the key challenge of selecting tiles with objects of interest is to *avoid any increase in the compute involved either on the camera side or on the server side*. We address this challenge of identifying tiles via a statistical approach that rests on a few observations. Our first observation is that since our cameras are stationary, only the objects of interest, such as vehicles and people, are in motion or can change. This implies that identifying just objects in motion is sufficient to identify objects of interest. Our second observation is that tiles' file sizes (or bitrates) increase significantly when there is any kind of motion in them. These two observations enable us to design a heuristic to identify tiles with objects of interest. Note that the tile bitrates are also influenced by other factors, such as weather conditions, the local conditions of the road, the color of the objects, and the position of the camera. Finally, we design a system TILECLIPPER that applies a threshold on the bitrates of the tiles seen in the past to identify the ones containing objects of interest. TILECLIPPER computes this threshold by performing a grid search on the statistics of tile bitrates during calibration. The grid search shows the right percentile values of the tile bitrates separately for the tiles with and without moving objects. It then uses the midpoint between these identified percentile values as the threshold. This threshold automatically adapts based on weather and traffic conditions for each tile independently.

We evaluate TILECLIPPER on a variety of videos by running it on embedded platforms like Jetson Nano and Raspberry Pi 4B (RPi). We show results of object detection as our system is agnostic to applications such as identification and tracking. Our dataset includes a total of 55 videos from standard benchmarks in various weather, lighting, and traffic density conditions and our own recorded videos. Our evaluation shows that TILECLIPPER achieves, on average, over 92% accuracy in identifying objects while running in real-time, even on RPi and Nano. This accuracy is higher than state-of-the-art frame filtering techniques like Reducto [38] and comparable to techniques like DDS [17], and CloudSeg [54] that use additional server computation. It also requires lower compute for calibration than the baselines, as it requires only one-time calibration instead of periodic re-calibrations. We summarize our contributions as follows:

- We identify that tiles with moving objects have higher bitrates and envision using them for traffic surveillance. To the best of our knowledge, this is the first work that uses tile bitrate to identify mobile objects.
- We utilize the correlation between tile bitrate and moving objects to design a thresholding strategy that identifies the tiles with moving/changing objects. This strategy is adaptive to different weather and traffic conditions.
- We evaluate TILECLIPPER on RPi and Jetson Nano and obtain $> 92\%$ accuracy. It works in real-time and reduces data sent by up to 40%. We compare it with DDS, Reducto, and CloudSeg to show that it is less compute-intensive, allowing scalable and cheaper deployment.
- We show a live deployment of TILECLIPPER near our university campus with a camera attached to an embedded board with a 4G connection. It saves over 55% bandwidth while providing an accuracy of over 87%.

2 Background & Motivation

In this section, we provide the background needed for our work and then present the motivation behind TILECLIPPER.

2.1 Working of Video Encoders

Videos are encoded typically in the form of a sequence of frames. To reduce the video's file size, it is generally not stored or encoded as raw pixels. Instead, each pixel is encoded by a reference to its neighboring pixels, where the neighbors can be drawn from either the adjacent pixels of the same frame (spatial dependence) or from adjacent frames (temporal dependence). Typically, an independent frame called an intra-frame is utilized after a fixed duration, followed by a sequence of inter-frames with content dependent on their previous frame. Such a sequence of intra followed by inter-frames is called a segment. A software called video decoder parses the dependencies within a segment to regenerate the actual content of the frames. The exact encoding scheme is specified by the video standard, also referred to as a codec.

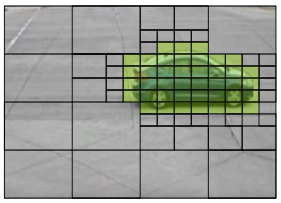


Figure 2: Video codecs use sub-blocks to encode complex scenes.

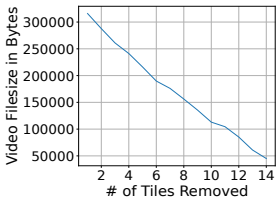


Figure 3: Removing tiles reduces the filesize of a video linearly.

We now discuss the strategy of video compression used by video codecs. In general, the video codecs operate at a coarser level than using pixel-level dependencies by partitioning the frames into spatial blocks (macroblocks in H.264 and coding tree units (CTUs) in H.265) [29]. Blocks with similar content (dependencies) can use the same bit allocation for representation, thus reducing the total number of bits needed. Matching blocks are estimated using motion vector predictor algorithms [50, 62]. Therefore, the objective of video encoders is to minimize the distortions (D) after compression and the number of bits (b) needed for encoding dependencies in the blocks (also referred to as rate R). This is known as R-D minimization. For frames split into K blocks, the objective function is shown mathematically as [45, 51]

$$\text{Minimize} \quad \sum_{k=1}^{K} (D(s_k, s'_k) + \lambda b_k), \tag{1}$$

where s_k is the original kth uncompressed block, s'_k is its decoded version of the compressed representation, $D(s_k, s'_k)$ is a distortion function that returns the mean squared error (MSE) between s_k and s'_k, λ is Lagrange multiplier assigning relative importance to distortion and the extra bits, and b_k is the number of extra bits needed to encode spatial (within frames) and/or temporal (across frames) dependencies of the kth block in a frame from a reference frame. In codecs such as HEVC and AV1, blocks with highly dynamic and complex scenes are further subdivided into smaller sub-blocks for precise motion prediction [50, 51]. This is shown in Figure 2 where blocks are subdivided into multiple levels to encode the area with moving car. Thus, the number of bits b_k for a block k and the total number of bits B for a segment with S total blocks that are subdivided using a single-level quadtree is given by:

$$b_k = \sum_{i=1}^{I} b_k^i, \text{ and } B = \sum_{k=1}^{S} b_k, \tag{2}$$

where b_k^i is the number of bits required for motion prediction of ith sub-block of kth block and I is the number of sub-blocks[2]. Note that while decoding the exact content requires algorithms, it is possible to obtain the total number of bits used in a segment B simply by parsing the statistics of the segments present in its headers.

[2]The number of sub-blocks depends on the codec used. For example, HEVC and AV1 use a quadtree structure with four recursive sub-blocks, whereas VVC uses a multi-tree structure

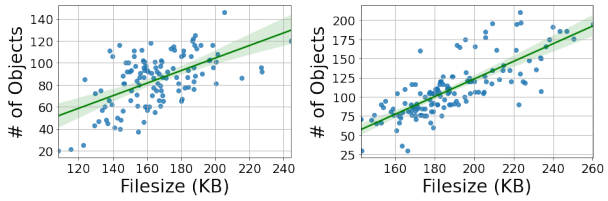


Figure 4: The plots show the correlation between video segment sizes and the number of moving objects for two different videos. The Spearman correlation values are 0.49 and 0.80, respectively.

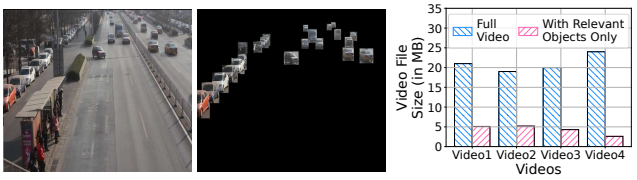
Tiles: The newer video standards such as HEVC, AV1, and VVC allow encoding videos in the form of independent rectangular units called *tiles*. Tiles allow individual spatial parts of frames to be encoded, removed, and decoded independently. Multiple works have, therefore, used tiles to optimize the delivery of 360 or omnidirectional videos [46, 58]. Note that tiles are different from blocks discussed in the above subsection. A tile consists of multiple blocks. During encoding, tiles are enforced with the constraint that the spatial and temporal dependencies among the blocks should be within blocks inside the tile boundaries only[41]. Therefore, the dependencies are localized and based only on the content within a tile.

Since tiles are independently encoded units of a video, removing them does not impact the decoder. The decoder can easily play the video by placing empty patches in the places where tiles are missing (Figure 1d). An important advantage of using tiles is that removing them reduces the video file size, thus saving network bandwidth. We confirm this in Figure 3 where we prune tiles one by one in 45 segments of a tiled video and observe that the average filesize of the segments reduces linearly. This implies that removing tiles with irrelevant scenes, such as the background with no objects, sky, trees, and buildings, can curb unnecessary bandwidth usage.

2.2 Correlation Between Number of Objects and Segment Bitrate

In surveillance settings, the camera itself is at a fixed location but needs to monitor any movement/changes. Due to the compression strategy of video codecs (discussed in §2.1), any sudden appearance and/or movement of objects would lead to a corresponding increase in the amount of residual data. Intuitively, this would lead to an increase in the bitrate (which effectively means filesize) of the video segments.

We first explain the intuitive reason behind the possible correlation between the number of moving objects and segment bitrates. Note from Eq. (2) and Figure 2 that regions of the video that have more motion and objects of interest are sub-divided into smaller blocks and thus require more number of bits to encode. Therefore, videos with moving objects and dynamic content have higher bitrates [15, 37]. Note that motion prediction and the use of smaller blocks for dynamic scenes are used by almost all modern video standards, such as H.264 [34], HEVC [24], AV1 [23] and VVC [27]. Such higher



(a) (b) (c)

Figure 5: Amount of possible savings in four sample videos. In (a) and (b), we show that the removal of the frame background leads to no loss of information. In (c), we quantify the possible data savings. Each video has a length of 1 minute and a resolution of 940×540. For "Full Video", we send the entire video unchanged. For "Relevant Objects", we remove the background.

bitrates have been observed by prior studies [15, 37, 60].

We confirm this hypothesis by studying the correlation between the filesize of videos and the number of objects, where the videos are segmented into lengths of 0.5s each. Figure 4 shows the scatter plots of two distinct videos in different settings (details of the benchmarks used are in §4). We note that the filesize and number of objects in the segments are moderately correlated, with the correlation values ranging from $0.48-0.90$. Such an observation motivates us to design a surveillance system that can use the bitrates to detect the presence of objects of interest, i.e., moving objects.

However, utilizing this observation to skip segments is difficult. This is because, as seen in Figure 4, no segment has zero objects, indicating that it is impossible to remove any segment without hurting the accuracy. Thus, an alternative strategy of pruning unnecessary regions is needed.

2.3 Shortcomings of Existing Systems

Recent works have proposed two major strategies to reduce the amount of data sent to the servers. The first strategy, frame-level filtering, is to transmit only a limited number of frames and then reconstruct the position of the moving objects from them [61]. The second strategy is to send only a low-resolution version of the video. The server side can then compensate for the low resolution by either running a more compute-intensive DNN [54] or, if needed, sending queries to the camera to send frames/regions of interest within frames at higher resolutions [17]. Figure 5 shows the amount of data that can be saved by sending only the relevant portions of the frame. We observe a reduction of $3.5-19\times$ in the filesizes that need to be sent if only the objects identified (known as regions of interest) are transmitted over the network.

However, sending only the regions of interest has a couple of challenges. First, traditional techniques of detecting objects require some type of DNN. Current techniques, such as DDS [17] and CrossROI [22], can only identify the spatial regions by running these compute-intensive DNNs. Although a few recent models like Yolo-Ret [20] can run in real-time on embedded devices with GPU, such as Jetson Nano, their reported accuracy is only comparable to the weakest full models (i.e., nano version of Yolov5).

While smart cameras, with specialized compute capabilities to run DNNs, are available in the market, they are at least $5\times$ costlier than conventional cameras [1]. Thus, current techniques either depend on server-side calibration heuristics as proposed by DDS [17] or Reducto [38] or utilize specialized hardware such as FPGA-based boards [21]. Second, pruning unnecessary regions or frames of interest often requires integration of the logic into the camera's firmware itself to avoid the overhead of re-encoding the video [38]. Both utilizing server-side techniques and integrating the logic into the firmware increase the complexity and cost of the system.

Next, we present TILECLIPPER that overcomes these limitations of prior works by using the key idea that bitrates and the presence of moving objects are correlated.

3 TILECLIPPER's Architecture & Design

3.1 Design Choices and Overview

To resolve the challenges discussed earlier, TILECLIPPER selects video tiles for transmission to the server in real-time. Unlike prior works, TILECLIPPER side-steps the problem of integration with firmware by using the tiled encoding built into the video standard. The main advantage of tiles is that removing some of them is not compute-intensive and can be done in real-time, even on embedded platforms. TILECLIPPER's identification of relevant tiles has the following motivations:

1. **Minimal computation requirement on the camera side:** The amount of computation on the camera side should be low enough to allow cheap edge devices like RPi to run the tile selection technique in real-time. This allows easier integration of the technique into existing deployments.
2. **Substantial bandwidth savings:** The number of tiles sent to the cloud server should be small enough to provide significant bandwidth savings.
3. **Sufficient robustness:** The selection of tiles should be robust enough so that server-side object detection accuracy does not suffer by missing out on necessary tiles under different light and weather conditions. The calibration of the system should not introduce a lot of additional computation overhead on the cloud/edge server.

TILECLIPPER Overview: The key idea behind TILECLIPPER is that video encoders utilize a technique of encoding that generates higher bitrates for tiles with moving or complex scenes than with simpler static scenes. This enables TILECLIPPER to identify the regions of interest without using the raw frames (i.e. the video is not decoded for TILECLIPPER's inference to run). This makes TILECLIPPER different from existing state-of-the-art techniques that often utilize optical flow [39] or low-level features of frames [38]. This technique

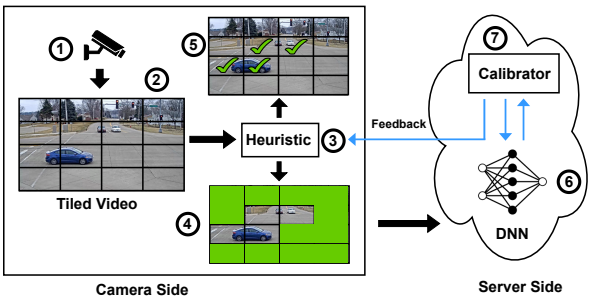


Figure 6: Workflow of TILECLIPPER. On the camera side, the camera generates tiled videos and runs the thresholding strategy to remove unwanted tiles. The server side runs the initial calibration and the DNN to recognize objects.

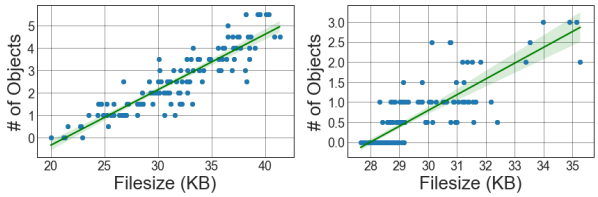


Figure 7: The plots show the correlation between video tile filesize and the number of objects for two different tiles. The Spearman correlation values are 0.78 and 0.90.

further has the advantage that it is easy to integrate TILECLIPPER with newer codecs, such as AV1 and VVC that utilize increasingly complex encoding and decoding algorithms.

Figure 6 shows the workflow of TILECLIPPER. At the camera side, ❶ camera hardware captures frames, ❷ provides the HEVC encoded tiled video, ❸ we apply TILECLIPPER's tile selection heuristic to select the relevant tiles from tiled video using a threshold. TILECLIPPER depends on a crucial observation to select tiles. We identify tile bitrate (in kbps) as a key statistic that can be used to identify whether there is a moving object in it. This is feasible because of the way video encoding schemes work in practice – where any static background material can easily be inferred via prediction from previous frames, but moving objects need to be encoded with additional bits [15, 37]. This increases the sizes of a group of tiled video frames with objects in them, as opposed to the ones that have only static background material. ❺ TILECLIPPER uses this insight to design a selection strategy. It calculates each tile's statistics separately and then individually identifies the right threshold for each of them by calibrating it (discussed in §3.3). ❹ Any tile that exceeds the threshold is sent to the server for detailed post-processing for traffic surveillance. ❻ At the server side, we use YOLO-v5 [8] for object detection and calibration. ❼ The calibrator uses the first 30*s* of videos and runs a DNN to send the right parameters as feedback to the camera. While we have assumed calibration to run on the server side, it is lightweight and rare enough to be performed even on edge devices with GPUs like nVidia Jetson Nano. Furthermore (as discussed in §5.2), calibration need not be triggered even by changes in weather/lighting conditions.

3.2 Utilization of Tile Bitrate Statistics

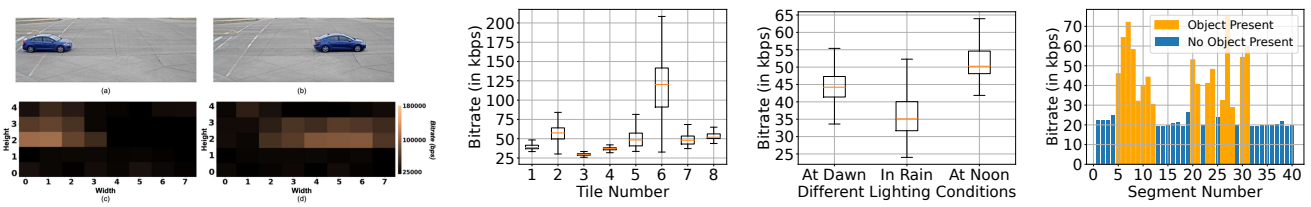
Correlation Between Tile Bitrate and Moving Objects: A major challenge of TILECLIPPER is to identify the spatial regions of interest. TILECLIPPER focuses on the smaller rectangular regions (tiles) as opposed to entire segments because there are few segments with no objects of interest. To utilize tiles, we first divide each segment into a configuration of 4×4 tiles, as justified by prior works such as CrossROI [22] since it gives a good balance between accuracy and overhead of encoding. We obtain the bitrates statistics of each tile and use a computer vision algorithm YOLO-v5 to obtain the number of objects in each of them (illustrated in Figure 1). A key thing to note is that obtaining the bitrates of each individual tile does not require decoding. Thus, this technique can be utilized on encoded videos by parsing only metadata.

We then obtain the correlation between the tile bitrates (filesizes) and the number of objects (shown for two tiles in Figure 7) for each individual tile. We find that there is a very strong correlation, with the Spearman correlation coefficient always exceeding 0.75. Figure 8a (for a subset of 32 tiles) shows only the tiles containing the moving car have higher bitrates. Note that even though the car is partly present in tiles, all of them show higher bitrates because the car is moving in all of these. Therefore, objects split across tiles show a similar signature in all tiles containing them. Such a strong correlation can be intuitively explained by the fact that since each tile is independently encoded, their sizes are affected by even a small number of moving/changing objects (see Figure 2). These changes cannot be as efficiently compressed by the encoding algorithm, leading to larger tile bitrates (in §2.1).

This strong correlation no longer holds if the comparison is performed across tiles with distinct scenes, even of the same segment (shown in Figure 8b). This is because the background encoded in each of the tiles affects the bitrates. We note that the tile bitrates have significantly different distributions, with the median values being different by a factor of 2.5×. Thus, each individual tile has its own bitrate characteristic that cannot be directly inferred from the adjacent tiles.

Lighting and Weather Conditions Affect Tile Bitrates: We also note in Figure 8c that both lighting and weather conditions affect tile bitrates. Again, this is intuitive since the background content can affect the tile bitrates. We find that in the rain, the droplets reduce the intensity of the colors, thus reducing the median tile size by over 30% compared to noon time. Since light and weather conditions can change periodically, these observations imply that the tile bitrate can be a good metric *only over the short term*. Thus, while tile bitrate can be used as a metric, the actual threshold should depend on a moving window of the bitrate statistics. An alternative strategy would be to recalibrate periodically, which has additional overheads (shown in Figure 16). However, we will show in §5.2 that using percentile statistics over a moving window performs as good as periodic recalibration.

Presence of Noise in Tile Bitrates: We next observe in Fig-



(a) (b) (c) (d)

Figure 8: (a) relation between the presence of moving objects and the bitrate (in bps) of tiles. Note that only the tiles where the car is moving have a higher bitrate than the other tiles. (b) and (c) shows bitrates of the same video across 600 segments of different tiles and the same tile under different lighting conditions, respectively. (d) shows the temporal (for 20s) distribution of video segments of a tile; spikes with high bitrates (in orange) have objects.

ure 8d that while the bitrates of a tile are noisy in nature, the amount of noise in general is small. The higher peaks usually represent the presence of moving objects, while the smaller peaks tend to represent minor changes, such as shadows, changes in lighting conditions, and so on. Therefore, just considering any tile bitrate higher than a certain arbitrary value does not necessarily imply the presence of an object. Thus, we discard smaller peaks by considering only the spikes as most probable to have objects with sizes greater than a threshold. Choosing this threshold via an intelligent strategy is, therefore, crucial as it is different for each tile and directly affects the accuracy and savings of TILECLIPPER. Moving objects of no interest, such as the movement of foliage, also gives a considerable spike in bitrate distribution. However, they should be discarded because they are irrelevant to traffic surveillance. Our calibration phases make the threshold high for these cases to reduce false detections.

3.3 Our Thresholding Technique to Filter Tiles

Problem Formulation: The goal of TILECLIPPER is to identify as many tiles with moving objects as possible while discarding tiles without such objects. The ratio of tiles selected with moving objects to the total tiles with moving objects is called recall. The ratio of tiles with objects present to the total tiles sent is called precision[3]. A high recall ensures that the most relevant objects are selected, whereas a high precision ensures that substantial bandwidth is saved. Let the bitrates of tiles $T_j (j = 1,\ldots,t)$ of a segment s_i be denoted as $B_i(T_j)$. Let their corresponding precision and recall be denoted by P_{ij} and R_{ij}, respectively. TILECLIPPER needs to identify the right threshold r_{ij} on bitrate $B_i(T_j)$ of each $< s_i, T_j >$ pair to classify if objects are present. An accurate classification would imply balancing both recall and precision. This is typically done using F_β-score, which is the weighted harmonic mean (HM) of precision and recall [40][4]. Mathematically,

$$\textbf{Maximize } F_\beta = \frac{1+\beta^2}{2} HM(P_{ij}, \frac{1}{\beta^2} R_{ij}), \forall s_i, T_j \qquad (3)$$

To assign the value of β, we note that recall is usually given higher priority than precision in traffic surveillance as it is often used in safety-critical applications [13]. In such cases, $\beta = 2$ is used as it gives $4\times$ higher weight to recall than to precision. We, therefore, maximize the F2-score. However, to explore the accuracy versus savings tradeoff for addressing use cases requiring higher precision, the value of 2 can be substituted by other values commonly used, such as 1 or 0.5.

Calibrating the Thresholds and Selecting Tiles: During the calibration, we send all the tiles of the first S segments to the server to run the *CALIBRATE* procedure. It starts by running the object classification (using YOLO-v5x) to identify tiles with moving objects inside by tracking bounding boxes using StrongSORT [19] object tracker in each frame of the segments (Line 2 of Algorithm 1). We use the intersection over union (IoU) of tracked bounding boxes to segregate tiles with static and mobile objects. We accordingly classify the tiles of the segments into two categories: *true* (with moving objects) and *false* (without moving objects). For each category, we compute the $10^{\text{th}}, 20^{\text{th}}, \ldots, 80^{\text{th}}$ percentiles[5] of the bitrates for each tile separately for all S segments, denoted by P_{10}^{true}, $P_{20}^{true}, \ldots, P_{80}^{true}$ and $P_{10}^{false}, P_{20}^{false}, \ldots, P_{80}^{false}$ respectively. We then consider the threshold as the mean of P_{v1}^{true} and P_{v2}^{false} for all values of $v1, v2 = 10$ to 80. We identify the pair of percentile values $v1$ and $v2$ using an exhaustive search that maximizes the F2-score for thresholding at the camera side (Line 4-8 in Algorithm 1). For each video, we use the first S=60 segments (30s) for calibration, which worked well for us during sensitivity tests (in Appendix A). After calibration, the server returns the best percentiles p_t and p_f and the clusters Q_t and Q_f with bitrates of the segments in S belonging to *true* and *false* categories, respectively, for each tile.

With the thresholds now calibrated, we use the *SELECT* procedure in Algorithm 2 at the camera to obtain the threshold on the bitrate. This is called for all the tiles of every segment. The procedure looks at the bitrates of the segments in each cluster (Q_t and Q_f) of each tile to compute the percentile values. To adapt the thresholds temporally, we maintain two different clusters of size 10 (sensitivity towards size shown

[3] We utilize precision and recall instead of the absolute number of true positives and true negatives as the data is often imbalanced in nature [40].

[4] Harmonic mean gives higher weightage to the term that has the least value. Thus, for harmonic mean, unlike in arithmetic or geometric mean, higher weightage to a term is given by dividing the term by the weight.

[5] We choose percentiles instead of weighted means due to its lower bias towards outliers, which are introduced by mistakes in the ground truth [56]. Furthermore, we utilize statistics instead of absolute values to adjust to content drift. We later show that this avoids the need for periodic re-calibration.

in the Appendix A) for each category. Note that each tile has its own independent cluster with ten recent bitrate values. We compare the bitrate with the tile-specific midpoint of $< P_{p\_t}^{true}, P_{p\_f}^{false} >$ (Lines 8-10 in Algo 2), where the values p_t and p_f are chosen during the calibration phase. The tile-specific clusters are updated in each call (Lines 12 and 14). A segment classified with no object is added to cluster Q_f, automatically removing the older element (because we use evicting queues[6] as clusters). A segment with objects is pushed to cluster Q_t. This update of clusters (an instance of a sliding window) ensures that the clusters are up to date with the changing bitrate distribution, making them adaptive to scene changes. Note that for tiles where we encounter no or very few segments with objects ($< 10\%$, i.e., < 6 segments) during calibration, this technique does not work because the clusters are not populated with enough values to represent the bitrate distribution. We fall back to an outlier detection approach for these cases (Lines 3-6).

Fallback to Outlier Detection: In a few tiles ($\approx 21\%$ of the total tiles in our dataset), the calibration phase finds no or too few segments with objects (no object ratio $O < 10\%$) within the S segments. These cases happen either because the tile focuses on an area outside the roads or because there is less traffic. We cannot decide to remove these tiles altogether because $\approx 26\%$ out of 21% (5.72% of the total) of tiles start having objects after calibration. Since these events are rare, we model them as the tail of a Gaussian distribution. To identify these, we compute the median P_{50} and the standard distribution σ of the bitrates of the tiles across past S segments, which is updated at each step (Line 6 of Algo 2). We then utilize $P_{50} + \gamma \times \sigma$ as the threshold (Line 5). To identify the right value of γ, we run TILECLIPPER on such tiles to get Figure 9 that shows how the false positives and misses vary with the increase in the value of γ. We choose $\gamma = 1.75$ (corresponding to the 96^{th} percentile), which gives a mean miss rate of < 0.05 and a mean false positive rate of ≈ 0.075. Note that we choose a slightly lower value of γ over the one that minimizes the total errors in Figure 9 for providing higher weightage to recall than precision.

Time Complexity of Calibration and Thresholding: We note that computing the percentile statistics of the tiles takes $O(S \log S)$, where S is the number of segments used for calibration. If there are t tiles, then this computation is repeated for each tile. Thus, calibration has a total time complexity of $O(tS \log S)$. For the SELECT procedure, again, we compute the percentiles of the two queues, which have a time complexity of $O(|Q| \log |Q|)$, where $|Q|$ is the maximum size of the two clusters (10 in our case). We also use enqueue operations, which take constant time. Since the SELECT procedure is called for all tiles t, this gives a total time complexity of $O(t|Q| \log |Q|)$. Since both t and Q are relatively small in magnitude (< 20 each), this time complexity is small enough for it to run at real-time on cheap devices.

[6] Evicting queues are FIFO queues with auto-dequeuing when enqueue exceeds the queue size.

Algorithm 1 TILECLIPPER's strategy for calibration

INPUT: Bitrate of each tile $B_i(T_j)$, list of percentile values V
OUTPUT: percentiles of true cluster and false clusters $< p\_t, p\_f >$, tiles in the cluster with objects Q_t, tiles in cluster with no objects Q_f

1: **procedure** CALIBRATE($B_i(T_j), V$)
2: Q_t, Q_f = Object recognition and tracking on S segments
3: $o \leftarrow [\,]$ // Empty list of objective values
4: **for** each value $v1$ in V **do**
5: **for** each value $v2$ in V **do**
6: $f2 \leftarrow$ ComputeF2Score($Q\_t, Q\_f, v1, v2$)
7: o.push($< v1, v2, f2 >$)
8: $< p\_t, p\_f > \leftarrow \arg\max_{<v1,v2>}(o)$
9: **return** $< p\_t, p\_f >, Q\_t, Q\_f$

Algorithm 2 TILECLIPPER's strategy to estimate the threshold for tiles of segments.

INPUT: Current tile's Bitrate B, no object ratio O, # used for calibration S, two evicting queues Q_t and Q_f containing bitrates of the past segments and best percentiles p_t and p_f to use for true and false clusters respectively, value of γ to use.
OUTPUT: Threshold on a tile.

1: **procedure** SELECT($Q\_t, Q\_f, O, S, B, p\_t, p\_f, \gamma$)
2: // Calculate threshold r
3: **if** $(sizeOf(Q\_t)/S) < O$ **then**
4: // Not enough objects; use fallback
5: $r \leftarrow (median(Q\_t) + \gamma * std(Q\_t))$
6: $Q\_t.enqueue(B)$ // Update cluster
7: **else**
8: $u \leftarrow percentile(Q\_t, p\_t)$
9: $l \leftarrow percentile(Q\_f, p\_f)$
10: $r \leftarrow 0.5 * (u + l)$ // Mean of both percentiles
11: **if** $B > r$ and $(sizeOf(Q\_t)/S) >= O$ **then**
12: $Q\_t.enqueue(B)$
13: **else if** $B <= r$ and $(sizeOf(Q\_t)/S) >= O$ **then**
14: $Q\_f.enqueue(B)$
15: **return** r

4 Implementation and Dataset

TILECLIPPER's Implementation: We implement both the camera-side version of TILECLIPPER and server side calibration in Python3. We obtain the bitrates of the tiled segments using the tool ffprobe available in the FFmpeg tool suite [4]. We run the camera-side component on Raspberry Pi 4 and Jetson Nano. Our server uses Ubuntu 18.04 and has an nVidia GeForce RTX 2080 Ti GPU. For the experiments, we store the encoded videos on the storage of Raspberry Pi or Jetson Nano and then use a script to start our workflow.

Video Encoding: While a number of traffic surveillance public datasets are available, they are usually not encoded in tiled form. Since TILECLIPPER requires the input video in tiled

Table 1: Summary of the dataset used for evaluation.

Dataset	# of Videos	Resolution	Duration	Type
AICC21	14	1920×1080	5 min	Benchmark
	14	1280×960		
DETRAC	20	960×540	1-2 min	Benchmark
Others	4	1280×720	6-8 min	Chaotic
OurRec	3	1280×720	13-25 min	Flyover
Total	**55**	-	-	-

form, we first need to encode the videos into tiled format using a system of 4×4 tiles. Other tiling configurations are possible, but we will discuss in §5.4 (as have prior works like CrossROI [22]) that this configuration performs well in practice. We first split the entire video into segments of $0.5s$ duration. We choose this duration since having it smaller wastes more bandwidth due to poor compression while increasing it reduces the scope of tile removal because fast-moving vehicles exit tiles quickly, leaving them empty mostly. We then encode it using the open-source HEVC encoder Kvazaar [53] at 30fps, and finally pack it into mp4 using the tool GPAC [36].

An encoding parameter that influences the quality of the videos is the Quantization Parameter (QP). A high QP value implies that both the amount of compression and loss is high. However, traffic surveillance videos do not require quality as good as in videos for end-consumers [17, 18]. Thus, we experimentally identify the QP parameter that leads to no loss of object detection accuracy using YOLO-v5s (in Figure 10). We observe that a QP parameter of 30 does not lead to any such loss while also reducing filesize by almost $4\times$, therefore we encoded all of our videos at 30 QP.

Datasets and Baselines: For large-scale evaluation, we use YOLO-v5s as the ground truth, as it is widely used for surveillance, by running it on each tiled video segment to identify objects. As in Reducto and DDS, we utilize publicly available benchmarks and a few of our own videos (Others and OurRec) in chaotic traffic and at flyovers (summarized in Table 1). The dataset has been carefully chosen to incorporate more hours of video and more diverse traffic conditions than the original studies that proposed the baseline strategies.

As a baseline, we re-implement DDS with a few changes in parameters to make the performance comparable to TILE-CLIPPER. We recall that DDS first sends the videos at a higher QP of 36 with lower $0.8\times$ resolution and identifies the regions with objects. The regions where objects are identified with low confidence are then queried from the camera to send at a lower QP than the original video (30 in our case with $0.8\times$ resolution). Our version of DDS uses HEVC video encoding and YOLO-v5s as the neural network on the server side (considering its state-of-the-art performance) [48], with a threshold value of $0.2 - 0.25$ to fetch an object for the second phase[7].

[7]A larger range of values leads to fewer misses and/or misclassifications, but at the cost of lower saving. We noted from experiments that increasing this range even by 0.05 leads to a large increase in objects recalled in the second phase, thus allowing very low or even negative savings compared to sending the whole video. This also highlights the critical dependency of DDS on DNNs' confidence score.

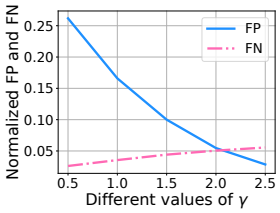


Figure 9: Increase in the value of γ decreases the chance of selecting tile falsely (FP) and increases the probability of misses (FN).

Figure 10: Varying video QP vs. filesize and YOLOv5 inference performance. YOLOv5 does not lose objects till 30 QP.

Although DDS always sends the video frames at a resolution $0.8\times$ the original, we still calculate the savings by comparing them with $1\times$ resolution to retain their setting and make the reported savings compatible with their implementation.

We utilize the open-source implementation of Reducto as a second baseline with fixes to circumvent the low QP video. Reducto's frame filtering uses a calibration system to first cluster the values of low-level pixel features. We reproduce the technique and then let the calibration run for $120s$, as we observed that this calibration time minimizes the need for re-calibration in the future.

Finally, we include two additional orthogonal baselines – CloudSeg and StaticTileRemoval (STR). We were able to run the open-sourced version of CloudSeg available at [43] without any changes. CloudSeg sends the videos at a lower resolution than the original and uses super-resolution at the server end before using object detection. We send the videos from the camera end at a resolution of $0.5\times$ the original and then use super-resolution at the server end for upscaling. In STR, we annotate the tiles that do not contain any road area and remove them. This allows us to test the utility of TILE-CLIPPER's thresholding strategy. We do not compare against AccMPEG [18] as its implementation requiring changes to the codec is currently not compatible with HEVC.

Performance Metrics: We report the following metrics:

(1) Accuracy: Accuracy is the ratio of objects that are detected after tile pruning to the total number of objects detected without filtering.

(2) Precision: Precision is the proportion of objects detected that are actually relevant (moving objects). A high precision indicates a substantial amount of bandwidth saving.

(3) Recall: Recall is the proportion of objects correctly identified. Having a high recall is essential to ensure that our system does not miss any moving objects.

(4) Bandwidth saving: The bandwidth saving in percentage is the amount of data reduction achieved as compared to the original data if the pruned video is sent.

(5) Execution time: We measure the execution time in frames per second on the camera side. We measure computations on the server side in terms of GPU use time in minutes.

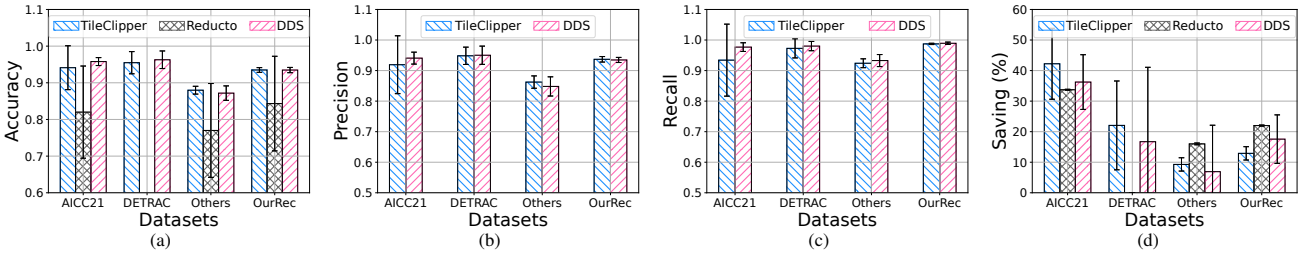


Figure 11: Performance of TILECLIPPER compared to Reducto and DDS in terms of (a) accuracy, (b) precision, (c) recall, and (d) bandwidth savings. TILECLIPPER provides accuracy, precision, and recall comparable to DDS, but larger bandwidth savings than DDS. TILECLIPPER provides much higher accuracy than Reducto (12-15%), though TILECLIPPER's savings are lower on average by 10%. We omit Reducto's precision and recall results as it never gives false positives, and also omit its results on DETRAC dataset as it requires more calibration time than the length of videos.

5 Evaluation

We evaluate TILECLIPPER's performance and justify some of its design choices discussed in §3.1 in various scenarios and compare it with DDS, Reducto, and CloudSeg.

5.1 Comparison with Baseline Techniques

Accuracy: Figure 11a compares the accuracy of TILECLIPPER and other baseline techniques across datasets. We list down our observations as follows: (1) TILECLIPPER is able to achieve a mean accuracy of over 0.85 on all the datasets. It exceeds 0.90 on AICC21, DETRAC, and OurRec. This indicates that very few moving objects are left undetected. (2) TILECLIPPER has comparable accuracy with DDS. Only on AICC21 videos DDS performs marginally better than TILECLIPPER. (3) Reducto suffers in terms of accuracy at testing time, leading to the lowest accuracy on all the datasets. Note that Reducto prepares a hash table of various thresholds, and the camera uses this threshold for filtering. Due to high quantization, the pixel level differences fail to be included within the existing clusters, leading to lower accuracy. We observe that this does not work well in practice on videos of lower quality, as corroborated by [52]. Further, we cross-verified Reducto's performance by running on videos with lower quantization to reproduce the accuracy originally reported.

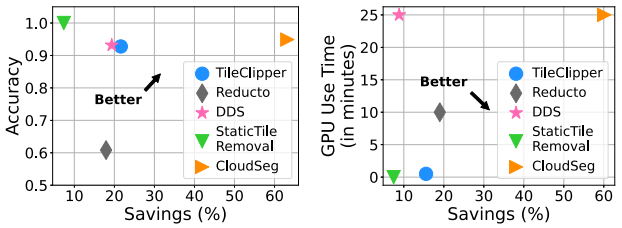
Precision and Recall: Figure 11b and 11c shows the precision and recall of TILECLIPPER and DDS. Since Reducto's frame filtration never provides false positives, we omit its computation of precision. We observe that TILECLIPPER provides > 0.85 precision and > 0.90 recall, which is high enough for most safety-critical applications [13]. DDS's both recall and precision are comparable to TILECLIPPER, thus proving that TILECLIPPER's threshold works well in selecting tiles. We further separately check the precision and recall values on the DETRAC dataset, as it contains such different traffic densities. We find that the fallback approach gives precision and recall values of 92.65% and 96.48%, respectively, which is only 2% lower than where normal calibration is possible.

Bandwidth Savings: Figure 11d shows a comparison of bandwidth saved in percentage for each technique. We observe that, TILECLIPPER provides $> 40\%$ savings for AICC21 and $> 20\%$ for DETRAC videos. For OurRec and Others, the saving is close to 10%. This is due to the nature of the videos because in these datasets, the number of cars and the traffic flow rate are very high and occupy most of the frame. This leaves less room for saving in terms of removing unnecessary tiles. Except for OurRec, on all other datasets, TILECLIPPER gives higher bandwidth savings than DDS. The savings for DDS are generally lower than TILECLIPPER even though it uses higher QP and $0.8\times$ video resolution. DDS also has a high variance compared to other techniques. This is attributed to the second phase of asking for the video at a high resolution if it has less confidence. Reducto has higher savings than both DDS and TILECLIPPER on all the datasets except on AICC21, where its savings are slightly lower. This is because Reducto's heuristic filters out frames even with dense traffic. These gains in bandwidth, however, come at the cost of accuracy, as shown in Figure 11a. Note that in cases where TILECLIPPER's accuracy and saving drop, the accuracy of DDS and Reducto drop too. This implies that the accuracy and saving drop is dataset specific and is not a shortcoming of TILECLIPPER.

Table 2: Comparison with CloudSeg and StaticTileRemoval (STR).

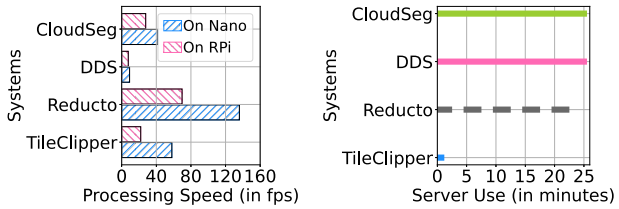
Dataset	Mean Accuracy (%)			Mean Saving (%)		
	Tile-Clipper	Cloud-Seg	STR	Tile-Clipper	Cloud-Seg	STR
AICC21	94.12	96.77	100	42.23	57.34	17.44
DETRAC	95.48	97.04	100	22.06	61.41	07.05
Others	87.98	90.80	100	09.28	69.89	00.00
OurRec	93.50	95.02	100	12.91	64.13	05.04

Comparison with Orthogonal Techniques: We also compare TILECLIPPER with the complementary works CloudSeg [54] and StaticTileRemoval (STR) in Table 2. STR removes the off-road tiles. Hence, none of the moving objects are missed, which leads to 100% accuracy at the cost of poor savings. Such savings even reach 0 on Others videos while TILECLIPPER provides 9.28% mean savings. Savings of TILECLIPPER is at least $2\times$ that of STR. CloudSeg provides the highest accuracy and saving, which is expected because it sends videos of half the resolution at the cost of increased server-side GPU use. TILECLIPPER's accuracies on AICC21, DETRAC, and OuRec are comparable to CloudSeg. In the case of OurRec and Others dataset, due to occlusions and



(a) (b)

Figure 12: (a) Accuracy and bandwidth saving tradeoffs of TILECLIPPER and baseline strategies across the entire dataset. (b) GPU use time and saving tradeoffs of all the systems for a video of ≈25 minutes.



(a) (b)

Figure 13: (a) shows mean video segment processing time (in fps) of TILECLIPPER, Reducto, DDS, and CloudSeg at the camera side on different hardware. (b) shows server use time of all the systems for a $\approx 25min$ video.

types of vehicles not present in the training dataset (COCO) of YOLO-v5 (our server-side DNN is trained on), TILECLIPPER's calibration misses objects leading to a drift from the optimal value. This makes TILECLIPPER filter tiles incorrectly. We leave utilizing a better server-side object recognition for future work. The adaptive spatial tile pruning of TILECLIPPER helps achieve high savings compared to StaticTileRemoval across the datasets.

Tradeoffs: Figure 12 shows the tradeoffs of all the techniques between accuracy, saving, and server-side GPU usage. We find that TILECLIPPER provides a better tradeoff between accuracy and bandwidth saving than the baselines (Figure 12a). CloudSeg gives the best tradeoff between accuracy and bandwidth saving at the cost of the highest server-side GPU use. At the server, as the majority of the cost comes from energy and money spent on GPU usage, we compare server-side GPU usage to process a $\approx$ 25 minute video against the saving in Figure 12b. Here, TILECLIPPER provides the best tradeoff between server-side GPU use and bandwidth savings. Although Reducto's savings are slightly better than TILECLIPPER, its GPU usage is 20$\times$ higher and gives the lowest accuracy. This implies that TILECLIPPER can maximize its accuracy and savings while using the least server-side resources.

Execution Time: We now test TILECLIPPER, Reducto, DDS, and CloudSeg benchmarks on two distinct edge devices – Jetson Nano and Raspberry Pi 4B (RPi). Figure 13a shows TILECLIPPER's processing speed along with the baselines on each device in terms of frames per second (fps) on the camera side. We note that TILECLIPPER runs at the rate of 22 fps on RPi, $\geq$ 16 fps [2], [10] needed for most real-time analytics, while on Jetson Nano, it runs at 57 fps. The higher speed on Jetson Nano is due to its support for hardware-based HEVC encoding on it. CloudSeg performs similar to TILECLIPPER on RPi because it also requires only video encoding. DDS is slower than TILECLIPPER on each device, as it requires extraction of the spatial portions from the frames (in its second phase), which is computationally expensive. Reducto runs fastest because it needs to encode fewer number of frames.

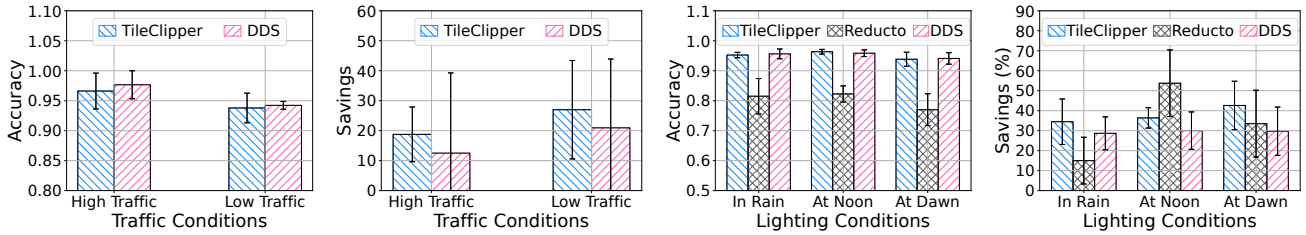
Cloud Server Cost: Figure 13b shows the server-side cost in terms of server GPU use time of each system to process a video of $\approx 25min$. We note that the most common object detectors available today are single-shot detectors, where the amount of computation is independent of the scenes. We observe that TILECLIPPER uses less computation (GPU) on the server side than others. It requires computation initially only during calibration for 60 video segments (30s). On the other hand, CloudSeg uses super-resolution continuously to scale up the low-resolution videos from the camera, and DDS requires processing all frames through a DNN to recall objects. Both are computationally expensive and use GPU to run super-resolution and object detection algorithms. In our experiments, Reducto required frequent recalibration every 2 minutes using the server's GPU. However, note that Reducto's computation for inference may also reduce slightly due to fewer frames eventually sent by the server. In summary, while TILECLIPPER runs in real time on edge devices and is computationally cheaper, it also uses less power (reducing costs) on the server side, making it an overall more efficient and scalable system.

5.2 Effect of Environment

Varying Traffic Density: Figure 14a and 14b show the accuracy and bandwidth savings achieved under different traffic densities. As expected, we note from Figure 14b that videos with less traffic density give the possibility of removing more tiles, resulting in more bandwidth savings. TILECLIPPER saves 22% more than DDS in both conditions. Furthermore, Figure 14a shows that TILECLIPPER works equally well in low traffic conditions, indicating that its fallback approach also gives good accuracy in practice.

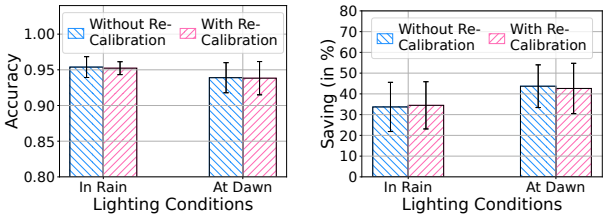
Varying Lighting Condition: We also have a set of three videos from the AICC21 dataset from the same camera with different lighting conditions – noon, rain, and dawn. We find that the accuracy values (shown in Figure 14c) do not change significantly. They are all more than 0.92 across all settings and comparable with DDS. Reducto performs worst, giving $<$ 0.85 accuracy in all the environments. TILECLIPPER's bandwidth savings (Figure 14d) are consistently $>$ 30% as compared to DDS in all the conditions, showing its robustness in all weathers. Reducto gives the highest savings only at noon, performing the worst in the rain because raindrops decrease the correlation between pixels of frames to filter.

Impact and Need for Recalibration: We argued in §3.3 that any change in weather and light conditions can be handled by TILECLIPPER's cluster-based adaptive threshold without recalibration. We confirm this in Figure 15a and 15b. We first



(a) (b) (c) (d)

Figure 14: Performance in different traffic densities and lighting conditions. (a) and (b) shows the accuracy and bandwidth savings under high traffic (>10 moving vehicles within a segment) and low traffic density. (c) and (d) shows the accuracy and bandwidth savings in the rain, at noon, and at dawn.



(a) (b)

Figure 15: Performance with and without re-calibration (a) shows accuracy and (b) shows bandwidth savings.

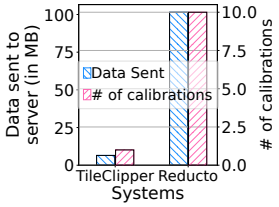


Figure 16: The amount of data sent to the server and the total number of recalibrations in TILECLIPPER and Reducto.

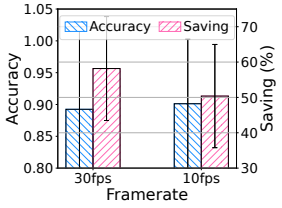
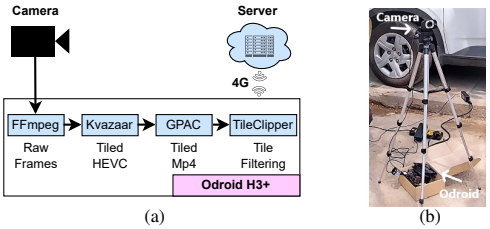


Figure 17: Accuracy and the bandwidth savings during the live experiment at different video encoding rates of 30fps and 10fps.

calibrate on the videos for noon and then run it both using re-calibration and without re-calibration for the videos in other conditions, such as rain and dawn. We note that recalibration has a negligible impact on accuracy and savings. This shows that TILECLIPPER's thresholding strategy of using percentile statistics of the clusters is robust enough to adapt to lighting and/or weather conditions, and additional recalibration is unnecessary to maintain consistent performance.

Recalibration Overheads: We now compare in Figure 16 the overheads involved in calibrating TILECLIPPER and Reducto in terms of data sent to the server during calibration and the total number of calibrations needed. We omit DDS and CloudSeg as they have no notion of calibration. We observe that for a 25*min* video, Reducto sends 15× more data to the server than TILECLIPPER for calibration. Further, the overall number of calibrations needed is also 10× more in the case of Reducto. This is because Reducto does not inherently adapt to scene changes and requires frequent recalibration triggers to get updated thresholds from the server. On the other hand, the adaptive algorithm (discussed in §3.3) of TILECLIPPER based on the clusters of the past 10 video segments can align itself with the temporal changes, making it more robust to changes in weather and traffic volumes changes, imposing lower server overheads.



(a) (b)

Figure 18: Live deployment, (a) Workflow on Odroid H3+, (b) On-site setup with a tripod-mounted camera connected to the Odroid.

5.3 Live Deployment of TILECLIPPER

We also evaluate TILECLIPPER in a live deployment where the key challenge was that surveillance cameras available today do not utilize tiled encoding. Live encoding in software, as done by Kvazaar, requires support for vector instructions that are unavailable on ARM processors. Thus, we designed a setup on Odroid H3+ [7], which had an Intel Jasper Lake N6005 x86 processor clocked at 2.0 GHz frequency. This costs around $130, which is similar to the cost of Jetson Nano.

Setup: Figure 18a shows the process of capturing frames, tiling, encoding, and running TILECLIPPER on top of it. To get 0.5 s of tiled video segments (needed for TILECLIPPER) at an fps of N, we first capture $N/2$ raw video frames from the camera using FFmpeg, encode them into HEVC tiled videos using Kvazaar (with a superfast encoding preset) and finally use GPAC to package them into tiled mp4. All these tools run in cascade and transfer data via Linux pipes on the Odroid H3+ board. We run the TILECLIPPER's thresholding algorithm on the encoded tiled video segments to discard tiles with no objects. We used a smartphone with 4G connectivity as a hotspot. After obtaining regulatory and IRB approval, we installed the setup about 4-5 meters from a two-way road intersection near our campus, having a vehicle speed limit of 30kmph. The traffic consisted of a mix of vehicles and pedestrians. We use a Logitech C615 1080p webcam as our camera connected to the Odroid H3+ board (shown in Figure 18b). Setting a bitrate filtering threshold in the deployment was similar to the design in §3.1. We sent the first 60 segments to our hosted server for calibration over the 4G network. Once the server returns the feedback, we let the tile filtering run. TILECLIPPER did not require any recalibration during the entire live experiment.

Results: We run two experiments of 30min and 15min dura-

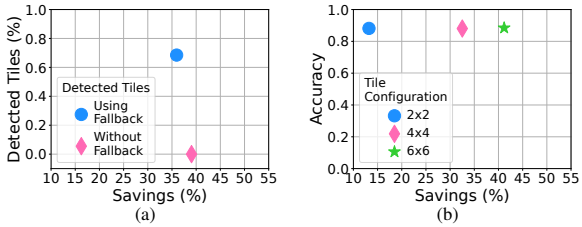


Figure 19: Sensitivity studies to (a) measure the effect of using fallback, and (b) show the effect of different tile configurations on accuracy and savings.

tion at frame rates of 30 and 10 fps, respectively. The accuracy and savings for both scenarios are reported in Figure 17. We observe that in this setup, TILECLIPPER obtains an accuracy $> 88\%$ and 90%, and saving $> 55\%$ and $\approx 50\%$ on 30 fps and 10 fps segments, respectively. However, even though the bandwidth saving in terms of percentage is similar, the amount of data sent is 53% higher when encoded at 30fps than at 10fps. We also observe that the streaming latency of filtered tiles is 50% and 30% lower after using TILECLIPPER compared to streaming the entire video at 30 fps and 10 fps, respectively. This shows that TILECLIPPER's bandwidth savings also reduce streaming latency, unlike DDS.

5.4 Sensitivity Study

We now justify the decision choices made in TILECLIPPER, including the parameters used during calibration and in the algorithmic strategies through sensitivity tests. We show additional sensitivity tests, namely, the need for a search of percentile statistics, the size of the clusters, and the calibration duration in Appendix A.

Effectiveness of Fallback: To validate the effectiveness of our fallback strategy for outlier detection (discussed in §3.3), we conducted a study with and without the fallback on 43 videos out of all 55 where the calibration phase identified no or too few segments with objects of interest (Figure 19a). Note that the no fallback approach leaves out all tiles, as it assumes that none of them would have objects. Thus, any strategy would show some fall in savings. TILECLIPPER's fallback strategy demonstrated a mere 8% reduction in savings compared to the scenario where it was not employed, as depicted in Figure 19a. Here, we show results for the tiles where clusters could not be formed during calibration due to the absence of objects. We observe that more than 65% of the tiles with objects were detected using the fallback strategy while having only 7% false selections. This implies that using our fallback strategy is more suitable than removing tiles altogether.

Using different tile configurations: We compare the accuracy with 4×4 tiles, among 2×2, 4×4 and 6×6 configurations in Figure 19b. In each case, the encoder uses constant-bitrate encoding to ensure that the size of the video does not change with tile configurations. While the accuracy remains relatively consistent irrespective of the setting used, we obtain substantial savings (over 30%) only with 4×4 and 6×6 configurations. While selecting 6×6 yields significant savings, it also escalates calibration complexities and increases the extraction time of bitrates by $3.5\times$ as opposed to 2×2 tiles. Thus, we selected the tile configuration of 4×4 since it strikes the optimal balance between savings and operational intricacies. A similar configuration was also chosen by CrossROI, citing an identical reason [22].

6 Related Works

Works on traffic surveillance fall into two categories – optimization of traffic surveillance using a variety of intelligent strategies and design of lighter DNNs.

Optimization of Traffic Surveillance: Traffic surveillance is increasingly used in practice, and thus is an important area of research [28, 32, 38, 44, 54, 55, 60]. Chameleon [32] and Spatula[31] use the traffic correlation from multiple cameras to identify segments to send to reduce bandwidth consumption. CASVA [60] uses a deep reinforcement learning mechanism to adapt the video parameters to the changing bandwidth configuration. AccMPEG [18] optimizes server-side DNN accuracy by tuning video codec-specific parameters to reduce bandwidth usage. This requires it to rely on the server side to get the right parameters. CloudSeg [54] sends the video at a low resolution but then uses super-resolution on the server side. Like CloudSeg, DDS [17] similarly also sends the video at a low resolution but then requests additional parts of frames separately when the DNN has low confidence. Clownfish [44] and AdaMask [39] extract the background content of the video frames and separately send only the objects to reduce the amount of data. However, identifying the objects requires a lightweight deep neural network, thus requiring a more expensive device on the camera side. Reducto [38] and SmartFilter [52] use a set of pixel-level operations to filter out irrelevant frames. MRIM [57] sends mixed-resolution frames having a higher resolution for areas with objects. Unlike TILECLIPPER, these works all work at the frame level, thus requiring the raw frames. Furthermore, like TILECLIPPER, CoVA [29] analyzes videos in the compressed domain to identify objects of interest. However, unlike TILECLIPPER, it uses neural networks on the server side.

Design of Lightweight Neural Networks: A number of works reduce the computation needed for vision tasks. For example, [25], [20] and [12] all run object detection on smaller edge devices like Jetson Nano. However, they still require a GPU for computation, and their speeds come at the cost of accuracy. FastDeepIoT [59] and DeepAdaptor [26] identify and then prune the nodes in the neural network for pruning to make it light enough for execution on embedded systems. NoScope [33], RECL [35] and Ekya [11] all design lightweight DNNs based on the situations observed by the cameras. Such techniques are orthogonal to our work.

7 Discussion on Design Choices, Limitations, and Future Work

We discuss a few aspects and limitations of TILECLIPPER:

Missing Tiles with Static and Small Objects: TILECLIPPER relies on the presence of moving objects to select tiles and, thus, misses static objects. As static objects would not raise the bitrate, the thresholding strategy would treat small movements as noise. Similarly, slow-moving objects could also be missed by TILECLIPPER, as they might not create sufficient changes in the bitrate. A possible mitigation strategy for slow-moving objects would be to use techniques like optical flow to interpolate missing frames/regions within videos. Since the changes in the case of slow-moving or static objects are relatively small, such interpolation can be done via optical flow estimation followed by pixel synthesis [16].

We also note that there is a slight drop in accuracy on TILECLIPPER with Others and OurRec datasets as they contain a few such missed cases. Furthermore, small objects ($< 1/20th$ of the tile dimensions), such as pedestrians or vehicles far from the camera do not increase the bitrate sufficiently to get detected by TILECLIPPER. They get considered as noise. However, missing such objects does not often hurt server-side application accuracy because such smaller objects ($< 1\%$ of the entire frame) are also missed by DNNs at the server.

Choice of Video Codecs in TILECLIPPER: We have evaluated TILECLIPPER on videos encoded using HEVC at constant bitrate (CBR). We choose HEVC as it is used in today's cameras [3, 5, 9], and there is open-source software available to manipulate its tiles easily. As newer codecs like AV1 and VVC support tiles and utilize similar encoding strategies, we expect TileClipper to also work with them. Note that H.264 encoders do not support tiled encoding, so it is difficult to use TILECLIPPER with H.264. Furthermore, we utilize CBR encoding as the tool used for video encoding, Kvaazar, does not support variable bitrate (VBR) encoding [6]. As VBR allows changes in bitrates of segments depending on the complexity of content, such changes in the individual tiles (utilized by TILECLIPPER) are even more strongly visible. Recent prior works on traffic surveillance such as AdaMask also utilized another similar technique called constant rate factor (CRF). Note that we do not consider adaptive bitrate (ABR), since none of the traffic surveillance systems currently use it. However, we believe that TILECLIPPER can still function by initially calibrating itself for each of the possible quality levels on the server.

Reduced Precision with Unstable Camera: The DETRAC dataset has a few videos with unstable cameras, causing movement across the frame, leading to lower precision and savings. We intend to explore reinforcement learning-based calibration to handle such problems by studying the correlation across tiles of segments. As the motion of moving objects on traffic would follow a pattern that leads to a temporal correlation among tiles. Such reinforcement learning-based camera calibration has shown good results in CamTuner [47].

Applying TILECLIPPER on Other Applications: Our evaluation discusses object recognition, because TILECLIPPER is agnostic to applications. This is unlike the baseline papers, such as DDS and Reducto, whose parameters need to change depending on the query sent to the camera. Because TILECLIPPER does not make any changes at the frame level or quality of videos, it has no direct effect on applications such as object classification, reidentification, or tracking. TILECLIPPER could also potentially be used to infer the level of congestion on roads since any change in the number and/or movement of objects in a tile would trigger TileClipper to send the content.

8 Conclusions

In this paper, we present a system TILECLIPPER for traffic surveillance that substantially reduces bandwidth consumption by selecting spatial regions (tiles) of interest for further analysis at the server. TILECLIPPER's tile selection leverages the observation that tile bitrates strongly correlate with the number of moving objects inside it. We then implement TILECLIPPER on inexpensive edge devices and show that it runs in real-time. We further show that it outperforms prior systems in accuracy, bandwidth savings and/or amount of computation in a wide range of scenarios and live deployment. We have made available our source code and datasets for use (in Appendix B) by the community to encourage further research.

Acknowledgments

This paper was partially funded by Cisco University Research Fund, Cisco Grant number 76417363 (SVCF Grant #2022-318921).

References

[1] Ambarella, AI Envisioned, Ambarella enables artificial intelligence on a wide range of connected cameras using Amazon SageMaker Neo. https://www.ambarella.com/news/ambarella-enables-artificial-intelligence-on-a-wide-range-of-connected-cameras-using-amazon-sagemaker-neo/. Visited on April 30, 2023.

[2] Average frame rate video surveillance statistics 2021. https://ipvm.com/reports/average-frame-rate-video-surveillance-2021. Visited on Oct 9, 2023; Published on Jan 8, 2021.

[3] *Dahua Technology, Network Video Recorder User's Manual*. Available at https://us.dahuasecurity.com/wp-content/uploads/2023/08/Network-Video-Recorder_Users-Manual_V2.3.1_20230210-Eng.pdf.

[4] Ffmpeg. https://www.ffmpeg.org/. [Online; accessed Sept-2021].

[5] GoPro Support, HEVC Explained. https://community.gopro.com/s/article/hevc. Published on Mar 22, 2023; Accessed on Aug 1, 2023.

[6] Is there any idea to achieve hevc variable bitrate mode in kvazaar encoder? https://github.com/ultravideo/kvazaar/issues/400. Published on Mar 21, 2024; Accessed on Jun 2, 2024.

[7] Odroid h3-plus. https://www.hardkernel.com/shop/odroid-h3-plus/. Accessed on May 1, 2023.

[8] Ultralytics yolov5. https://github.com/ultralytics/yolov5. Accessed on August 1, 2022.

[9] *Wisenet Network Camera: User Manual*. Available at https://www.hanwhasecurity.com/wp-content/uploads/attachments/u/s/user_manual-xnp-6120h-english_web-0710.pdf.

[10] IBM Documentation, Camera frame rate, resolution, and video format requirements. https://www.ibm.com/docs/en/video-analytics/1.0.6?topic=requirements-camera-frame-rate-resolution-video-format, March 2021.

[11] Romil Bhardwaj, Zhengxu Xia, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, Nikolaos Karianakis, Kevin Hsieh, Paramvir Bahl, and Ion Stoica. Ekya: Continuous learning of video analytics models on edge compute servers. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 119–135, Renton, WA, April 2022.

[12] Yuxuan Cai. *YOLObile: Real-time object detection on mobile devices via compression-compilation co-design*. PhD thesis, Northeastern University, 2020.

[13] Andrea Ceccarelli and Leonardo Montecchi. Evaluating object (mis)detection from a safety and reliability perspective: Discussion and measures. *IEEE Access*, 11:44952–44963, 2023.

[14] Chacha Chen, Hua Wei, Nan Xu, Guanjie Zheng, Ming Yang, Yuanhao Xiong, Kai Xu, and Zhenhui Li. Toward a thousand lights: Decentralized deep reinforcement learning for large-scale traffic signal control. *AAAI Conference on Artificial Intelligence*, 34(04):3414–3421, Apr. 2020.

[15] Jan De Cock, Zhi Li, Megha Manohara, and Anne Aaron. Complexity-based consistent-quality encoding in the cloud. In *2016 IEEE International Conference on Image Processing (ICIP)*, pages 1484–1488, 2016.

[16] Jiong Dong, Kaoru Ota, and Mianxiong Dong. Video frame interpolation: A comprehensive survey. *ACM Transactions on Multimedia Computing and Communication Applications*, 19(2s), May 2023.

[17] Kuntai Du, Ahsan Pervaiz, Xin Yuan, Aakanksha Chowdhery, Qizheng Zhang, Henry Hoffmann, and Junchen Jiang. Server-driven video streaming for deep learning inference. In *2018 Conference of the ACM Special Interest Group on Data Communication*, page 557–570, New York, NY, USA, 2020.

[18] Kuntai Du, Qizheng Zhang, Anton Arapin, Haodong Wang, Zhengxu Xia, and Junchen Jiang. Accmpeg: Optimizing video encoding for accurate video analytics. In *Proceedings of Machine Learning and Systems*, volume 4, pages 450–466, 2022.

[19] Yunhao Du, Zhicheng Zhao, Yang Song, Yanyun Zhao, Fei Su, Tao Gong, and Hongying Meng. Strongsort: Make deepsort great again. *IEEE Transactions on Multimedia*, 2023.

[20] Prakhar Ganesh, Yao Chen, Yin Yang, Deming Chen, and Marianne Winslett. YOLO-ReT: Towards high accuracy real-time object detection on edge GPUs. In *IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*. IEEE, 2022.

[21] Shikha Goel, Rajesh Kedia, M Balakrishnan, and Rijurekha Sen. Infer: Interference-aware estimation of runtime for concurrent cnn execution on dpus. In *International Conference on Field Programmable Technology (ICFPT)*, 2020.

[22] Hongpeng Guo, Shuochao Yao, Zhe Yang, Qian Zhou, and Klara Nahrstedt. Crossroi: Cross-camera region of interest optimization for efficient real time video analytics at scale. In *12th ACM Multimedia Systems Conference*, page 186–199, 2021.

[23] Jingning Han, Bohan Li, Debargha Mukherjee, Ching-Han Chiang, Adrian Grange, Cheng Chen, Hui Su, Sarah Parker, Sai Deng, Urvang Joshi, Yue Chen, Yunqing Wang, Paul Wilkins, Yaowu Xu, and James Bankoski. A technical overview of AV1. *Proceedings of the IEEE*, 109(9):1435–1462, 2021.

[24] Philipp Helle, Simon Oudin, Benjamin Bross, Detlev Marpe, M. Oguz Bici, Kemal Ugur, Joel Jung, Gordon Clare, and Thomas Wiegand. Block merging for quadtree-based partitioning in hevc. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1720–1731, 2012.

[25] Rachel Huang, Jonathan Pedoeem, and Cuixian Chen. Yolo-lite: A real-time object detection algorithm optimized for non-gpu computers. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 2503–2510, 2018.

[26] Yakun Huang, Xiuquan Qiao, Jian Tang, Pei Ren, Ling Liu, Calton Pu, and Junliang Chen. Deepadapter: A collaborative deep learning framework for the mobile web using context-aware network pruning. In *IEEE Conference on Computer Communications*, 2020.

[27] Yu-Wen Huang, Jicheng An, Han Huang, Xiang Li, Shih-Ta Hsiang, Kai Zhang, Han Gao, Jackie Ma, and Olena Chubach. Block partitioning structure in the vvc standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 31(10):3818–3833, 2021.

[28] C. Hung, G. Ananthanarayanan, P. Bodik, L. Golubchik, M. Yu, P. Bahl, and M. Philipose. Videoedge: Processing camera streams using hierarchical clusters. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 115–131, 2018.

[29] Jinwoo Hwang, Minsu Kim, Dohee Kim, Daeun Kim, Seungho Nam, Yoonsung Kim, Hardik Sharma, and Jongse Park. CoVA: Exploiting Compressed-Domain Analysis to Accelerate Video Analytics. In *2022 USENIX Annual Technical Conference*.

[30] Md Rokebul Islam, Nafis Ibn Shahid, Dewan Tanzim ul Karim, Abdullah Al Mamun, and Md Khalilur Rhaman. An efficient algorithm for detecting traffic congestion and a framework for smart traffic control system. In *2016 18th International Conference on Advanced Communication Technology*. IEEE, 2016.

[31] Samvit Jain, Xun Zhang, Yuhao Zhou, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, Paramvir Bahl, and Joseph Gonzalez. Spatula: Efficient cross-camera video analytics on large camera networks. In *ACM/IEEE Symposium on Edge Computing*, 2020.

[32] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. Chameleon: Scalable adaptation of video analytics. In *2018 Conference of the ACM Special Interest Group on Data Communication*, page 253–266, 2018.

[33] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. Noscope: Optimizing neural network queries over video at scale. *Proceedings of VLDB Endowment*, 10(11):1586–1597, Aug 2017.

[34] Chao-Yang Kao and Youn-Long Lin. A memory-efficient and highly parallel architecture for variable block size integer motion estimation in h. 264/avc. *IEEE transactions on very large scale integration (VLSI) systems*, 18(6):866–874, 2009.

[35] Mehrdad Khani, Ganesh Ananthanarayanan, Kevin Hsieh, Junchen Jiang, Ravi Netravali, Yuanchao Shu, Mohammad Alizadeh, and Victor Bahl. RECL: Responsive Resource-Efficient continuous learning for video analytics. In *20th USENIX Symposium on Networked Systems Design and Implementation*, pages 917–932, Boston, MA, April 2023.

[36] Jean Le Feuvre. GPAC filters. In *Proceedings of the 11th ACM Multimedia Systems Conference*, pages 249–254, Istanbul Turkey, May 2020. ACM.

[37] Weihe Li, Jiawei Huang, Shiqi Wang, Chuliang Wu, Sen Liu, and Jianxin Wang. An apprenticeship learning approach for adaptive video streaming based on chunk quality and user preference. *IEEE Transactions on Multimedia*, 25:2488–2502, 2023.

[38] Yuanqi Li, Arthi Padmanabhan, Pengzhan Zhao, Yufei Wang, Guoqing Harry Xu, and Ravi Netravali. Reducto: On-camera filtering for resource-efficient real-time video analytics. In *Annual Conference of the ACM Special Interest Group on Data Communication*, page 359–376, 2020.

[39] Shengzhong Liu, Tianshi Wang, Jinyang Li, Dachun Sun, Mani Srivastava, and Tarek Abdelzaher. AdaMask: Enabling Machine-Centric Video Streaming with Adaptive Frame Masking for DNN Inference Offloading. In *Proceedings of the 30th ACM International Conference on Multimedia*, pages 3035–3044. ACM, October 2022.

[40] Christopher D Manning. *An introduction to information retrieval*. Cambridge university press, 2009.

[41] Kiran Misra, Andrew Segall, Michael Horowitz, Shilin Xu, Arild Fuldseth, and Minhua Zhou. An Overview of Tiles in HEVC. *IEEE Journal of Selected Topics in Signal Processing*, 7(6):969–977, December 2013.

[42] Saman Naderiparizi, Pengyu Zhang, Matthai Philipose, Bodhi Priyantha, Jie Liu, and Deepak Ganesan. Glimpse: A programmable early-discard camera architecture for continuous mobile vision. In *15th Annual International Conference on Mobile Systems, Applications, and Services*, page 292–305, 2017.

[43] Kyung-Ah Sohn Namhyuk Ahn, Byungkon Kang. Fast, Accurate, and Lightweight Super-Resolution with Cascading Residual Network. https://github.com/nmhkahn/CARN-pytorch.

[44] Vinod Nigade, Lin Wang, and Henri Bal. Clownfish: Edge and cloud symbiosis for video stream analytics. In *ACM/IEEE Symposium on Edge Computing*, 2020.

[45] Jens-Rainer Ohm, Gary J. Sullivan, Heiko Schwarz, Thiow Keng Tan, and Thomas Wiegand. Comparison of the Coding Efficiency of Video Coding Standards—Including High Efficiency Video Coding (HEVC). *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1669–1684, December 2012.

[46] Sohee Park, Arani Bhattacharya, Zhibo Yang, Samir R. Das, and Dimitris Samaras. Mosaic: Advancing user quality of experience in 360-degree video streaming with machine learning. *IEEE Transactions on Network and Service Management*, 18(1):1000–1015, 2021.

[47] Sibendu Paul, Kunal Rao, Giuseppe Coviello, Murugan Sankaradas, Oliver Po, Y. C. Hu, and Srimat Chakradhar. Enhancing video analytics accuracy via real-time automated camera parameter tuning. In *20th ACM Conference on Embedded Networked Sensor Systems*, 2023.

[48] Maxim Priymak and Richard Sinnott. Real-Time Traffic Classification through Deep Learning. In *2021 IEEE/ACM 8th International Conference on Big Data Computing, Applications and Technologies (BDCAT '21)*, pages 128–133, Leicester United Kingdom, December 2021. ACM.

[49] K. K. Santhosh, D. P. Dogra, and P. P. Roy. Anomaly detection in road traffic using visual surveillance: A survey. *ACM Computing Surveys*, 53(6), dec 2020.

[50] Gary J. Sullivan, Jens-Rainer Ohm, Woo-Jin Han, and Thomas Wiegand. Overview of the High Efficiency Video Coding (HEVC) Standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1649–1668, December 2012. Conference Name: IEEE Transactions on Circuits and Systems for Video Technology.

[51] G.J. Sullivan and R.L. Baker. Rate-distortion optimized motion compensation for video compression using fixed or variable size blocks. In *IEEE Global Telecommunications Conference GLOBECOM '91: Countdown to the New Millennium. Conference Record*, pages 85–90, Phoenix, AZ, USA, 1991. IEEE.

[52] Jude Tchaye-Kondi, Yanlong Zhai, Jun Shen, Dong Lu, and Liehuang Zhu. Smartfilter: An edge system for real-time application-guided video frames filtering. *IEEE Internet of Things Journal*, pages 1–1, 2022.

[53] Marko Viitanen, Ari Koivula, Ari Lemmetti, Arttu Ylä-Outinen, Jarno Vanne, and Timo D. Hämäläinen. Kvazaar: Open-source hevc/h.265 encoder. In *24th ACM International Conference on Multimedia*, 2016.

[54] Yiding Wang, Weiyan Wang, Junxue Zhang, Junchen Jiang, and Kai Chen. Bridging the edge-cloud barrier for real-time advanced vision analytics. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[55] Yiding Wang, Weiyan Wang, Junxue Zhang, Junchen Jiang, and Kai Chen. Bridging the edge-cloud barrier for real-time advanced vision analytics. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.

[56] Rand R Wilcox and HJ Keselman. Modern robust data analysis methods: measures of central tendency. *Psychological methods*, 8(3):254, 2003.

[57] Ji-Yan Wu, Vithurson Subasharan, Tuan Tran, and Archan Misra. MRIM: Enabling Mixed-Resolution Imaging for Low-Power Pervasive Vision Tasks. In *2022 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 44–53, Pisa, Italy, March 2022. IEEE.

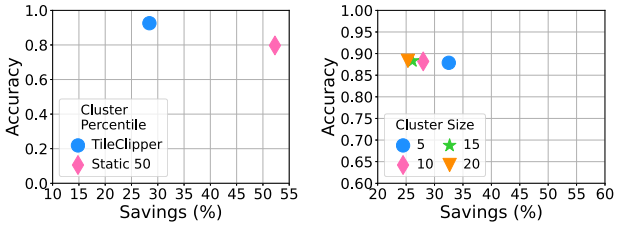
[58] Praveen Kumar Yadav and Wei Tsang Ooi. Tile rate allocation for 360-degree tiled adaptive video streaming. In *28th ACM International Conference on Multimedia*, page 3724–3733, 2020.

[59] Shuochao Yao, Yiran Zhao, Huajie Shao, ShengZhong Liu, Dongxin Liu, Lu Su, and Tarek Abdelzaher. Fastdeepiot: Towards understanding and optimizing neural network execution time on mobile and embedded devices. In *16th ACM Conference on Embedded Networked Sensor Systems*, page 278–291, 2018.

[60] Miao Zhang, Fangxin Wang, and Jiangchuan Liu. Casva: Configuration-adaptive streaming for live video analytics. In *IEEE Conference on Computer Communications*, pages 2168–2177, 2022.

[61] Tan Zhang, Aakanksha Chowdhery, Paramvir Bahl, Kyle Jamieson, and Suman Banerjee. The design and implementation of a wireless video surveillance system. In *21st Annual International Conference on Mobile Computing and Networking*, pages 426–438, 2015.

[62] Yongfei Zhang, Chao Zhang, Rui Fan, Siwei Ma, Zhibo Chen, and C.-C. Jay Kuo. Recent Advances on HEVC Inter-Frame Coding: From Optimization to Implementation and Beyond. *IEEE Transactions on Circuits and Systems for Video Technology*, 30(11):4321–4339, November 2020.



(a) (b)

Figure 20: Sensitivity to different parameters of TILECLIPPER on a subset of the videos. (a) shows the effect of choosing a single fixed percentile for all tiles vs grid searched value of TILECLIPPER. (b) shows the effect of different cluster sizes on accuracy and savings.

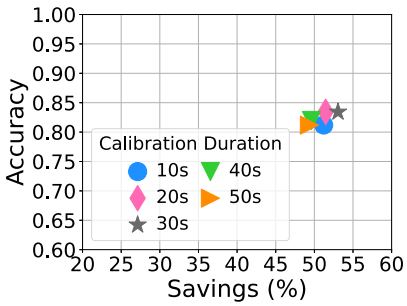


Figure 21: Effect of varying calibration duration on TILECLIPPER. We choose 30*s* as it gives the best tradeoff between accuracy and savings.

A Supplementary Materials Appendix

A.1 Additional Sensitivity Results

We now discuss three additional sensitivity results to justify the search for tile statistics, the size of clusters, and the duration of calibration used in TILECLIPPER. The study utilizes the same dataset as in §5.4.

Without Search of Tile Statistics: We first recall that TILECLIPPER uses a search of the percentile statistics to find out the optimal one. To verify that this search is essential, we perform our study with a threshold value fixed at 50^{th} percentile for both the clusters across 8 videos, selecting 2 videos from each of the 4 datasets randomly. This results in more savings ($\approx 2\times$) at the cost of lower accuracy when compared with the study run on the same dataset but with calibration as illustrated in Figure 20a. This happens due to the static nature of the threshold value. The tiles with objects of interest having a lower bitrate than the fixed threshold are missed, affecting accuracy. Our selection of 50^{th} percentile is deliberate, as opting for any other lower value would have led to a compromise in accuracy.

Varying cluster sizes: In Figure 20b, we show experimentation with different cluster sizes. Interestingly, we find that varied cluster sizes do not affect accuracy. However, we observe a decrease in savings by 3.34% when cluster size changes from 10 to 15. It is important to note that we avoid a cluster size of 5 over the choice of 10 as it captures the underlying bitrate distribution with a mere 7.5% dip in savings. Furthermore, we also note that choosing any size above the cluster size of 10 leads to a progressive decline in savings.

Varying calibration duration: Figure 21 demonstrates the calibration phase across different time durations, we deliberately chose videos from the dataset where the Tileclipper exhibited its worst performance. The motivation behind our experiment is to observe the effect of varied calibration duration. We observe that changing the calibration duration has relatively small effects on the overall accuracy and savings. Since increasing the calibration time did not contribute to an improvement in accuracy or savings instead, we chose to utilize an average calibration of 30s to reduce the incidents of fallback.

B Artifact Appendix

We now describe the technique of reproducing the results discussed in the evaluation of TILECLIPPER. This open-sourced artifact provides the required codebase to run TILECLIPPER along with documentation to reproduce the results reported in this paper.

B.1 Scope

The artifact consists of all the required pre-processed datasets to reproduce the results quickly. In addition, we provide sample videos/datasets to validate TILECLIPPER and the baselines.

B.2 Contents

The file *README.md* in the *main* branch describes the right steps and procedures for evaluation. Note that the codes require the dataset to be downloaded and placed in the appropriate folder before evaluation.

B.3 Hosting

All the source codes of TILECLIPPER are open-sourced on GitHub at https://github.com/shubhamchdhary/TileClipper. The datasets can be downloaded from Zonodo at https://doi.org/10.5281/zenodo.11179900.

B.4 Requirements

Although TILECLIPPER is evaluated on a diverse set of hardware (like Raspberry Pi, nVidia Jetson Nano, and Odroid H3+), it can also be run on any Linux-based (tested on Ubuntu 20.04) computer with nVidia GPU. It depends on a few frameworks/tools that can be easily built using the instructions in the repository. The detailed requirements are specified in the README file.