VIDEOCHEF: Efficient Approximation for Streaming Video Processing Pipelines

Ran Xu[α], Jinkyu Koo[α], Rakesh Kumar[α], Peter Bai[α], Subrata Mitra[β]
Sasa Misailovic[γ], Saurabh Bagchi[α]
α: Purdue University, β: Adobe Research, γ: University of Illinois

Abstract

Many video streaming applications require low-latency processing on resource-constrained devices. To meet the latency and resource constraints, developers must often approximate filter computations. A key challenge to successfully tuning approximations is finding the optimal configuration suited for content characteristics, which are changing across and within the input videos. Searching through the entire search space for every frame in the video stream is infeasible, while tuning the pipeline off-line, on a set of training videos, yields suboptimal results.

We present VIDEOCHEF, a system for approximate optimization of video pipelines. VIDEOCHEF finds the optimal configurations of approximate filters at runtime, by leveraging the previously proposed concept canary inputs (using small inputs to tune the accuracy of the computations and transferring the approximate configurations to full inputs). VIDEOCHEF is the first system to show that canary inputs can be used for complex streaming applications. The two key innovations of VIDEOCHEF are (1) an accurate error mapping from the approximate processing with downsampled inputs to that with full inputs and (2) a directed search that balances the cost of each search step with the estimated reduction in the run time.

We evaluate our approach on 106 videos obtained from YouTube, on a set of 9 video processing pipelines (in total having 10 distinct filters). Our results show significant performance improvement over the baseline and the previous approach that uses canary inputs. We also perform a user study that shows that the videos produced by VIDEOCHEF are often acceptable to human subjects.

1 Introduction

Video processing has brought many emerging applications such as augmented reality, virtual reality, and motion tracking. These applications implement complex video pipelines for video editing, scene understanding, object recognition and object classification [14, 49]. They often consume significant computational resources, but also require short response time and low energy consumption. Often, the applications need to run on the local machines instead of the cloud, due to latency [14], bandwidth [50], or privacy constraints [46].

To enable low-latency and low-energy video processing, we leverage the the fact that most stages in the video pipeline are inherently *approximate* because human perception is tolerant to moderate differences in images and many end goals of video processing require only estimates (*e.g.*, detecting object movement or counting the number of objects in a scene [5]). Many *domain-specific* algorithms have exposed algorithmic knobs, that can e.g., subsample the input images or replace expensive computations with lower-accuracy but faster alternatives [45, 40, 13, 25]. To complement domain-specific approximations, researchers have proposed various generic *system-level techniques* that expose additional knobs for optimizing performance and energy of applications while trading-off accuracy of the results. The techniques span compilers [38, 27, 41, 7, 34], systems [3, 15, 18, 17, 28], and architectures [30, 26, 35, 34, 8].

Content-dependent Approximation. A fundamental challenge of uncovering the full power of both generic and domain specific approximations is finding the configurations of these approximations that provide maximum savings, while providing acceptable results for *each given input*. This challenge has two main parts.

First, the optimal approximation setting is dependent on the *content of the video*, not just on the algorithms being used in the processing pipeline. Often individual videos, or parts of the same video should have different approximation settings, requiring the program to make the decisions at runtime. Second, the optimization needs to explore a large number of approximate configurations before selecting the optimal one for the given input, re-

quiring the optimizer to construct off-line models. Systems like Green [3] and Paraprox [34] dynamically adapt the computation using runtime checks of the intermediate results, while Capri [42] selects approximation level from the input features at the program start. However, the systems rely on extensive off-line training to map the approximation levels to accuracy and performance.

To relax the dependency on off-line training, Laurenzano *et al.* [24] propose Input Responsive Approximation (IRA) for runtime recalibration with no offline training. IRA creates *canary inputs*, smaller representations of the full inputs (obtained via subsampling), and then reruns the computation on the canary input with different approximation settings, until it finds the most efficient setting that maintains the accuracy requirement (on the canary). While the concept is promising, the application of IRA to video processing pipelines is limited:

- IRA has been applied to individual computational kernels (in contrast to full pipelines). It is unclear how to capture the interactions between the stages of the pipeline, how often to calibrate, and what are the optimal canary sizes.
- IRA uses the approximation settings derived from the canary input to the full input, assuming that the errors for the full and correlated inputs will be *similar*. However, the assumption is often incorrect (98% of cases, Figure 2) and leads to missed speedup opportunities.
- IRA's greedy search may introduce additional overheads and may not find good approximation settings efficiently because it has no notion of what are the appropriate points in the stream to search.

Our Solution: VIDEOCHEF. We present VIDEOCHEF, a fast and efficient processing pipeline for streaming videos. VIDEOCHEF can optimize the performance subject to accuracy constraints for the system-level and domain-specific approximations of all kernels in the video processing pipeline. Figure 1 presents VIDEOCHEF's end-to-end workflow:

- Like IRA, VIDEOCHEF uses small-sized canary input to guide the the on-line search for approximation setting. However, unlike IRA, VIDEOCHEF is tailored for optimization of the whole video processing pipelines, not just individual kernels.
- In contrast to IRA, VIDEOCHEF presents a finely tunable prediction model for mapping the error from the canary input to that with the original input. This prediction model is trained offline and hence does not generate any additional runtime overhead. At the same time, it is much more lightweight than the full off-line training employed by other approaches.
- At runtime, VIDEOCHEF performs an efficient search through the space of approximation settings and ensures that the cost of the search does not overwhelm the benefit of approximating the computation.

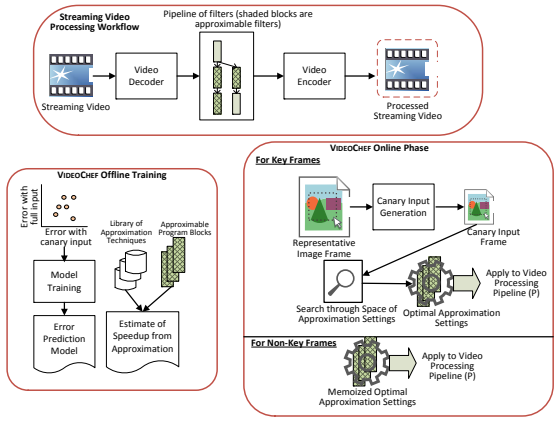


Figure 1: End-to-end flow of approximate video processing with VIDEOCHEF. The video processing pipeline comprises multiple filters, which can be approximated to save computation at the expense of tolerable video quality. The offline and the online components of VIDEOCHEF work together to determine the best approximation setting for each approximable filter block.

We evaluate VIDEOCHEF with three error models and two search strategies, applied to a corpus of 106 YouTube videos from 8 content categories, which span the range of video features (e.g., color and motion). We analyze 10 filters arranged in 9 pipelines of size 3. We find that VIDEOCHEF is able to reach within 20% of the theoretical best performance possible and outperforms IRA's performance by 14.6% averaged across all videos and saves on an average 39.1% over the exact computation given a relatively restrict quality requirement. While given a more loose quality requirement, VIDEOCHEF is able to reach within 26.62% of the theorectical best one and also achieve higher performance gain – 53.4% and 61.5% over IRA and exact computation, respectively.

While we have framed this discussion in terms of video processing, the novel contributions outlined below apply to other low-latency streaming applications, with the fundamental requirement that the characteristics change to some extent from one segment of the stream to another, for instance, online video gaming, augmented reality and virtual reality applications.

Contributions. We make the following contributions:

- We present VIDEOCHEF, a system for performance and accuracy optimization of video streaming pipelines. It consists of off-line and on-line components, that together adapt the application's approximation level to the desired output quality.
- We build a predictive model to accurately estimate the quality degradation in the full output from the error generated when using the canary input. This enables more aggressive approximation setting to the approximation algorithm that has tunable knobs.

- We propose an efficient and incremental search technique for the optimal approximation setting that takes hints from the video encoding parameters to reduce the overhead of the search process.
- We demonstrate the benefits of VIDEOCHEF through (1) quantitiative evaluation on various real-world video contents and filters and (2) a user study.

2 Background and Motivation

Error Metric: At a high-level, a video is composed of a sequence of image frames. To quantify the error in the output or the processed video due to approximation, we measure the Peak Signal-to-Noise Ratio (PSNR) of the output video. PSNR is the average of the PSNRs of the individual frames in the output video. Suppose that a video consists of K frames where each frame has $M \times N$ pixels. Let $Y_k(i,j)$ be the value of the pixel at (i,j) position on the k-th frame of the processed video without any use of approximation, and $Z_k(i,j)$ be the value of the pixel when approximation was applied. Then, the PSNR of the approximate output is computed as follows:

$$PSNR = \frac{1}{K}\sum_{k=0}^{K-1} 20 \times log_{10}\frac{MaxValue}{\sqrt{MSE(Z_k, Y_k)}}, \quad (1)$$

where $MaxValue$ is the maximum possible pixel value present in the frame, and $MSE(Z_k, Y_k)$ is the mean square error between Z_k and Y_k, i.e., $\sum_i \sum_j (Z_k(i,j) - Y_k(i,j))^2$, as a result of approximation. Thus, lower the PSNR, the higher the error in the output video.

Isn't the problem solved by IRA and Capri? IRA (Input Responsive Approximation) [24] and Capri [42] attempted to address the problem of selecting optimal approximation level for individual inputs.

IRA [24] solely relies on canary inputs to search for best approximation settings. Thus, it implicitly assumes that the magnitude of error corresponding to a particular approximation setting on the *canary inputs* is identical to the error with the same approximation settings on the *full-sized inputs*. But, Figure 2 shows our experiment with 424 real images and 216 different approximation settings. We found that for the same approximation settings, the PSNR of the full-sized inputs can be significantly different from the PSNR of the canary inputs. Most of the points (about 98%) are above the diagonal, indicating that the error on the full input is lower than that with the canary input for the same approximation level.

We attribute the difference in the approximation to the higher variations between neighboring pixel values for canary inputs. Therefore, for the same approximation settings, the approximate processing on canary inputs gives lower PSNR. We found that on an average, the PSNR of a full-sized output is 5.36 dB higher than the PSNR of canary output. Therefore, IRA misses an opportunity for more aggressive optimization that can fit within the user-specified quality threshold.

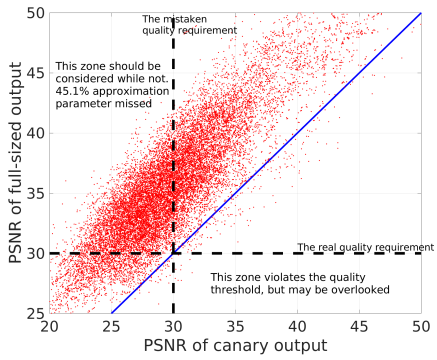


Figure 2: The PSNR of full-sized output versus the PSNR of canary output, for the I-frames of 106 videos on one of our application Boxblur-Vignette-Dilation video filter pipeline. The PSNR of full output is higher for over 98% approximation settings and 45.1% of the approximation settings lie in such a zone that is ignored by IRA approach but actually satisfies the quality requirement.

Capri [42] rigorously addresses the problem of selecting the best approximation settings to minimize the computational cost, while meeting the error bound. But Capri also fails in the video processing setting because it does not recalibrate itself with the stream and thus cannot change its approximation settings when the characteristics of the stream change. Further, it (1) relies on prior enumeration of all possible inputs, which is impossible in this target domain, and (2) performs the selection of approximation settings completely offline, which reduces the cost of the optimization but makes it non-responsive to changes in the input data.

3 Solution Overview

Figure 1 shows the end-to-end workflow of streaming video processing, with approximation.

Approximation. Under normal processing, a video decoder converts the video into its constituent frames. Then a sequence of "filters" (synonymously, processing steps or pipeline stages) is applied to each frame. Examples include blurring filter (*e.g.*, at the TSA airport checkpoint scanners) and edge detection filter (*e.g.*, for counting people in a scene). Finally, the transformed frames are optionally put together by a video encoder. To make such processing fast and resource efficient, VIDEOCHEF intelligently uses selective approximation (Section 3.1) during the computation of the filters. The user sets the quality constraint on the output video quality. An example specification is that the PSNR of the output video should be above 30 dB.

Accuracy Calibration with Canary Inputs. For each representative frame (called "key frame" here), VIDEOCHEF determines a *canary input* (Section 3.2),

which summarizes the full frame such that the dissimilarity between the full and the canary frame remains below a threshold. With the canary input, VIDEOCHEF occasionally recalibrates the approximation levels of the filters. It determines when to call the search algorithm using domain specific knowledge about the frames and scenes (Section 3.3). For this, we extract hints from the video decoder which lets VIDEOCHEF determine the key frames. This amortizes the cost of the search across many frames of the video, with 80-120 frames being a typical range for MPEG-4 videos. In the absence of such a video decoder, we have a variant, which triggers the search upon a scene change detection.

Online Search for Optimal Tradeoffs. VIDEOCHEF searches for the approximation setting of each filter that gives the lowest execution time subject to a threshold for the output quality (Section 3.4). Since the search for approximation is done with the *canary input*, the error of approximate computation is different from the error of the computation on the full input. VIDEOCHEF introduces a method to accurately map between these two errors (Section 3.5). In performing this estimation, we consider multiple variants of VIDEOCHEF, depending on what features are available to the predictor, such as, some categorization of the video frame according to its image properties. Through this procedure, we aim to maximally leverage the approximation potential in the application and give flexible approximation choices.

3.1 Approximation techniques

The computations involved in filtering operations can be approximated by VIDEOCHEF in various ways as long as each of the underlying approximation techniques exposes knobs that can be tuned to control the approximation levels (ALs). For example, in the three popular program transformation-based approximation techniques, the variable `approx_level` is a tuning knob that controls the levels of approximation. A higher value implies more aggressive approximation, leading most often to higher speedup but also higher error. These transformations are performed automatically by a compiler (LLVM in our case).

Loop perforation: In loop perforation [41, 27], the computation is reduced by skipping some iterations, as shown below.

```
for (i = 0; i < n; i = i + approx_level)
  result = compute_result();
```

Loop truncation: In loop truncation [41, 27], the last few iterations of the computation are dropped as shown in the following example:

```
for (i = 0; i < (n - approx_level); i++)
  result = compute_result();
```

Loop memoization: In this technique [7, 34], for some iterations in a loop we compute the result and cache it. For other iterations we use the previously cached results.

```
for (i = 0; i < n; i++)
  if (i % approx_level == 0)
    cached_result = result = compute_result();
  else result = cached_result;
```

3.2 Canary Inputs

To reduce the search overhead for finding the best approximation level within each frame of the video, we generate canary inputs for the frame following the work in [24]. A good canary input should meet two requirements: (1) it should be *close enough* to the original input so that the AL found by the canary is the same as the AL computed from the original; (2) it should be small enough that the search process using the canary input is efficient. We first define the dissimilarity metric to compare the canary sample video and the full-sized video and then show how to choose the appropriate canary input.

Metrics of Dissimilarity. We define two metrics of dissimilarity. Let a full-sized video have K frames and $M \times N$ pixels in each frame, and each pixel has the property $X(i,j)$. A canary video has K frames with $m \times n$ pixels, and the same property $Y(i,j)$. The property could be one component in the YUV colorspace of an image, where the Y component determines the brightness of the color (known as "luminance") while the U and V components determine the color itself (known as "chroma") and each ranges from 0 to 255. The "dissimilarity metric for mean" (SMM), is defined as follows (following [24]):

$$mFull = \frac{1}{M \times N \times K} \sum_{i=0}^{K-1} \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} X(i,j) \tag{2}$$

$$mSmall = \frac{1}{m \times n \times K} \sum_{i=0}^{K-1} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} Y(i,j) \tag{3}$$

$$SMM = \frac{|mSmall - mFull|}{mFull} \tag{4}$$

When a pixel has a vector of values, such as the YUV colorspace which has 3 values for the 3 components, then the SMM metric is combined across the different elements of the vector. The combination could be a simple average or a weighted average; we use the latter due to the higher weightage of the Y-channel in the YUV colorspace. Similarly, we define the "dissimilarity metric of standard deviation" (SMSD) to capture the dissimilarity in the Standard Deviation between the full input and the canary input.

Generating Candidate Canary Videos. Given a frame of the video from which to generate the canary video, we resize the frame to a fraction $1/N$ of its original size to create the canary video. Typical sizes that we find useful in our target domain are $1/16$, $1/32$, $1/64$, $1/128$, $1/256$

of the original size. Since the frame is a 2-D matrix of pixels, to resize it to $1/N$ of its original size, we shrink the width and height each to $1/\sqrt{N}$ of the full size by sub-sampling 1 pixel out of every $\sqrt{N}$ pixels.

Reducing an input size causes at least proportional reduction in the amount of work inside the filter. Many filters that are finding increasing use are super-linear, where the benefit of using a small canary frame is even more significant. Two popular examples are determining optical flow to measure motion [4] and morphological filter [10], where the value of each pixel in the output image is based on a comparison of the corresponding pixel in the input image with its neighbors. We compute the similarity between the full-sized video frame and the canary video frame according to the metrics SMM and SMSD. We set the maximum dissimilarity metric we can tolerate as a threshold parameter—we find 10% is a practically useful threshold for both SMM and SMSD. Among all the qualified canary inputs, we select the smallest one as our final choice.

3.3 Identifying Key Video Frames

Searching for the best AL for each approximable program block is computationally expensive. Conceptually, we would want to repeat the search when the characteristic of the video changes significantly so that the optimal approximation setting is expected to be different. In practical terms, we want to perform such change point detection without having to parse the content of the video. Video encoders already provide hints when the content of the scene has changed significantly.

We make the observation that videos have temporal locality, and many frames in the *same group* will have the same approximation setting. Therefore, we can perform a single search once per the group of frames, once we identify the group. We leverage domain-specific knowledge about videos to automatically select the group boundaries in two ways:

Scene Change Detector. Our first observation is to recalibrate the approximation at the beginning of different scenes. This approach is general and works for any video format. There are mainly 2 classes of scene change detectors, namely, pixel-based and histogram-based. The pixel-based methods are highly sensitive to camera and object motion. Histogram-based methods are good for detecting abrupt scene changes. To keep our overhead low (since the detection algorithm runs on every frame), we limit ourselves to detecting only abrupt scene changes and use canary frames for this detection.

We implement a histogram-based scene change detector using only the Y-channel of frames [20]. We experimentally found the Y-channel information was sufficient to detect abrupt scene changes and we were more concerned about overhead of scene change detector than its accuracy. The algorithm detects a scene change whenever the sum of the absolute difference across all the bins of histograms of two consecutive frames is greater than some predefined threshold (20% of the total pixels in our evaluation).

I-frame Selection for MPEG videos. The second solution takes advantage of I-frames, present in the popular H.264 encoder (which the MPEG-4 and many other video formats follow). It defines three main types of frames: $I-$, $P-$, and $B-frames$ [21]. An I-frame uses intra-prediction meaning the predicted pixels within this frame are formed using only samples from the same frame. The P- and the B-frames use inter-prediction meaning the predicted pixel within this frame depends on samples from the same frame *as well as* samples from other frames around it (the distinction between P- and B-frames is not relevant for our discussion).

When to insert an I-frame (also called a "reference frame") depends on the exact coding scheme being used, but in all such coding schemes that we are aware of, a big difference in the frame triggers the insertion of a new I-frame, since inter-coding will give almost as long a code as intra-coding. Further, because an I-frame does not have dependencies on other frames, this makes it easier to reconstruct and perform the (exact or approximate) computation. We see empirically that for a wide range of videos used in our evaluation, the average spacing between adjacent I-frames is 137 frames. Although specific to only some video formats, it results in a low sampling rate and consequently the low search overhead, while triggering search at a suitable granularity.

3.4 Search with Canary Inputs

An approximable program block exposes one or more approximation knobs. The `approx_level` variable mentioned with the loop-based approximation techniques in Section 3.1 is an example of such a knob. In our notation, we use AL 1 to denote the exact computation. The higher the AL is, the less accurate the computation is and the higher the speedup is. Now for a pipeline of cascaded filters, each having one or more approximation knobs we have a *vector* of approximation settings per frame. We define a *setting* in the processing pipeline as the combination of ALs for each of the approximable program blocks in the video processing pipeline. For example, with an n-stage processing pipeline and each stage being approximable and having exactly one approximation knob, the setting will be $\vec{A} = \{a_1, a_2, \cdots, a_n\}$, where a_i denotes the AL of knob i.

To find the best approximation setting, we follow a searching algorithm outlined as follows:

1. **Start searching** at a particular setting, typically $(1, 1, \cdots, 1)$, corresponding to no approximation.

2. **Select a group of candidate settings** by the Candidate Selection Algorithm. The selection algorithm simply selects the next set of settings to try out in the search process. The greedy algorithm works as follows: given the current setting $\vec{A^{(0)}}$, we assume that we can reach the best AL by looking at each step 1 AL further in each approximable block. So the candidates are $\{\vec{A^{(i)}}\}$, with $i = 1 \cdots n$, for n approximable blocks and $\vec{A^{(j)}} = \{a_1^{(0)}, \cdots a_{j-1}^{(0)}, a_j^{(0)} + 1, a_{j+1}^{(0)}, \cdots, a_n^{(0)}\}$.
3. **Decide whether to continue search**, i.e., whether it is worthwhile to try any of these candidate settings. We use the Approximation Payoff Estimation Algorithm (Section 3.4.1). If not, return the current setting. This algorithm estimates whether the saving due to the more aggressive approximation can compensate for the time of the additional search step.
4. **Try each worthwhile candidate setting** from the set computed by the previous step. Use the ALs in candidate settings to run approximate computation on canary video and compute the error metric for each candidate setting. Then map the error metric to that with the full sized outputs.
5. **Check for exit or iterate** – if error metrics of all the full video outputs exceed the error boundary, return the current setting. Otherwise, the candidate setting which gives the lowest error becomes the next setting, go to (2) and iterate.

3.4.1 Approximation Payoff Estimation Algorithm

The goal of this algorithm is to estimate the benefit of executing the application with the new AL searched for versus the cost of searching with the new AL, all for the key frame under question. Let the current setting be represented by $\vec{A^{(0)}} = \{a_1^{(0)}, a_2^{(0)}, \cdots, a_n^{(0)}\}$. Recollect that this is the set of ALs for each of the approximable blocks in the application. Let the execution time of the application at setting $\vec{A^{(i)}}$ and with canary downsampling C_d be given by $g(A^{(i)}, C_d)$, where $C_d = 1$ denotes the execution time with the full input.

This algorithm works in a breadth-first fashion and attempts to prune some of the paths where exploring higher degrees of approximation for a particular knob cannot speedup the execution further and may lead to slowdown due to associated overheads. From the current setting of $\vec{A^{(0)}}$, let the next possible settings of exploration be $\vec{A^{(1)}}, \vec{A^{(2)}}, \cdots, \vec{A^{(N)}}$. For example, with greedy search, with n approximable blocks, there will be n possible next settings. The maximum possible benefit by exploring all the candidate next settings is calculated as:

$$B = \max_{i=1}^{N} [g(A^{(0)}, 1) - g(A^{(i)}, 1)] \quad (5)$$

This benefit B simply means the maximum reduction in execution time across all the possible candidate settings, when run with the full input. However, to realize this gain, we have to pay the cost of searching, which can be expressed in terms of the overhead as follows:

$$O = \sum_{i=1}^{N} g(A^{(i)}, C_d) \quad (6)$$

This overhead O is simply the cost of executing the application with the next step ALs, but with the canary input (and hence the downsampling ratio C_d). The decision for VIDEOCHEF becomes simple: if $B > O$, then continue the search, else stop and return the current setting.

3.5 Error mapping model

We have to develop an error mapping model to characterize the relation between error in the canary output and error in the full output, for the same approximation levels. This is important because we have seen empirically (Figure 2) that the canary errors are higher than full frame errors for most points. We propose three different mapping models to use according to different amounts of knowledge in the model.

3.5.1 Model-C

Suppose we know the error metric of a canary output C. The error metric of a full-sized output F is estimated by a quadratic regression model as follows,

$$F = w_0 + w_1 \times C + w_2 \times C^2 \quad (7)$$

Offline, we calculate the ground truth of the pairs (C, F) for every possible AL $\vec{A}$ and for all the videos in the training set. In practice, we find that sub-sampling the space of possible ALs still provides accurate enough training, with a sub-sampling rate of 10% being adequate. Let us say that the error bound specified by the user is E_B. Then clearly we want $F < E_B$. However, due to the possible inaccuracy of the error mapping model, we want to explore a larger space so that we are not missing out on opportunities for approximating. Therefore, while training the model, Model-C, we explore all the points where $F \leq E_B + \Delta$, where Δ is a user configurable parameter for how far outside the tolerable region we want to explore in the model. Then we solve the unknown coefficients w_0, w_1, and w_2 in the model. We find empirically that for a large set of videos, this model reaches its limits with the quadratic regression function.

3.5.2 Model-CA

Now, suppose VIDEOCHEF has additional knowledge of what ALs were in effect. Given the error metric of a

canary output C, which is computed approximately with ALs $\vec{A} = \{a_1, a_2, \cdots, a_n\}$, we construct the input vector $\vec{I} = (1, C, \vec{A})$. The first element of this vector is the constant 1 and allows for a constant term in our equation for F. Then the error metric of a full-sized output F is estimated by a regression model as follows,

$$F = \vec{I} \cdot w \tag{8}$$

where w is a $(n+2) \times 1$ coefficient matrix. The goal of the training is to estimate the matrix w. The elements of the matrix w provide the weights to multiply the different input components—the error in canary output (C), and the different ALs. We use similar training method offline as for Model-C.

3.5.3 Model-CAD

Many of approximation techniques on image processing reduce computation load by skipping a fraction of rows of images. Thus, the difference over rows is often related with approximation quality. Inspired by this characteristic, we consider a new feature vector $\vec{D} = (d_1, d_2, d_3)$, where each of d_k's represents a feature extracted from one of Y, U, and V channels of an image. The feature d_k is referred to as a row-difference feature and is defined as the mean of absolute difference in pixel values of the same column between consecutive rows in each channel. Averaging over rows and columns, we use only one representative number as d_k for each channel.

Considering an input vector $\vec{I} = (1, C, \vec{A}, \vec{D})$, the error metric of a full-sized output F is estimated by a linear regression model as:

$$F = \vec{I} \cdot w, \tag{9}$$

where w is a $(n+5) \times 1$ coefficient matrix. In the experiment results, we will see that Model-CAD outperforms the other models.

3.5.4 Non-linear models.

We have also tested complex non-linear models to predict F, using artificial neural networks with all pixel information as input. However, considering the run-time complexity, we could not observe any significant benefit of the non-linear models over the linear models mentioned earlier. Thus, we do not report their results in the evaluation.

4 Implementation and Dataset

We use loop perforation and memoization [41, 27] to approximately filter the frames in the video. The implementation of VIDEOCHEF is comprised of an offline and an online component. The offline component uses a set of training videos (50% of videos described under the dataset below) and creates models for the error mapping and for the cost and the benefit of each step of the search. This last model is actually implemented as a lookup table, due to the space being only piece-wise continuous. During runtime, VIDEOCHEF queries these models, using linear interpolation if needed, and performs an efficient search to identify the optimal ALs and runs each of the three filters in any pipeline with their optimal values.

VIDEOCHEF API. Our compiler pass identifies the approximable blocks using program annotations and then performs the relevant transformations to insert the approximation knobs to be tuned (such as `approx_level` in Sec. 3.1). The user can then use the following API calls to enable VIDEOCHEF in the video pipeline:

- *setCalibrationFrequency(f="I-frame")* : This will set how frequently VIDEOCHEF will search for the best approximation settings. The default value is VIDEOCHEF will trigger a search for every I-frame. If *f*="*x*", then VIDEOCHEF will search every *x*-th frame.
- *setQualityThreshold(b="30")* : This will set the (lower) PSNR threshold that the approximated pipeline must deliver. Default is 30 dB. VIDEOCHEF exposes to the user approximate versions of many filters from the FFmpeg library, with names like *deflate_approx*. The developer of VIDEOCHEF can register a callback with the video decoder using the call *void notifyIFrame(void *).*

Video Dataset. We used 106 YouTube MPEG-4 videos for our evaluation. We used `libvideo`, a lightweight .NET library [23], to download the videos. The videos were collected from 8 different categories to cover a spectrum of different motion and color artifacts in the frames: Lectures, Ads, Car Races, Entertainment, Movie trailers, Nature, News, and Sports. At the first step, a single seed video was downloaded from each category, then we downloaded all YouTube's recommendations to the seed video, which turned out to belong to the same category as that of the seed video. Once the set of videos was collected, we randomly sub-sampled a 20 second clip from each video, being motivated by a desire to bound the experiment time. For each category, we collected approximately 25 videos and filtered out those with low resolution (since the quality threshold was likely already breached with the original video).

5 Evaluation

We describe our benchmarks first and then the four experiments to evaluate the macro properties of VIDEOCHEF and then its various components.

Table 1: Summary of the analyzed pipelines. We denote the approximation applied to each filter: Loop Perforation (LP) or Memoization (M)

Name	Description, labeled with Approx. type	Approximation Type	Approximation Levels
DEB	Deflate(LP)-Emboss(LP)-Boxblur(M)	Loop perforation(LP) & Memoization(M)	1-6, 1-6, 1-6
DVE	Deflate(LP)-Vignette(LP)-Emboss(LP)	Loop perforation	1-6, 1-6, 1-6
BVI	Boxblur(M)-Vignette(LP)-Inflate(LP)	Loop perforation & Memoization	1-6, 1-6, 1-6
UIV	Unsharp(LP)-Inflate(LP)-Vignette(LP)	Loop perforation	1-6, 1-6, 1-6
DUE	Dilation(LP)-Unsharp(LP)-Emboss(LP)	Loop perforation	1-6, 1-6, 1-6
BVD	Boxblur(M)-Vignette(LP)-Dilation(LP)	Loop perforation & Memoization	1-6, 1-6, 1-6
UEE	Unsharp(LP)-Erosion(LP)-Emboss(LP)	Loop perforation	1-6, 1-6, 1-6
EUB	Erosion(LP)-Unsharp(LP)-Boxblur(M)	Loop perforation & Memoization	1-6, 1-6, 1-6
BUC	Boxblur(M)-Unsharp(LP)-Colorbalance(LP)	Loop perforation & Memoization	1-6, 1-6, 1-6

Benchmarks. We construct our benchmark by including different video processing pipelines. Each video processing pipeline consists of 3 consecutive filters, which are selected from a pool of 10 video filters from the FFmpeg library. These filters are modified to support approximation with tuning knobs. To execute on these filter pipelines, one needs to provide a video input and a quality threshold. Finally, the output is also a video, together with a quality metric with respect to each frame. We have a total of 9 different filter pipelines.

Quality Metric. We use PSNR (Eq. 1) as the quality metric for the videos produced by the approximate pipelines. We present the results for two acceptable PSNR thresholds. The threshold of 30 dB is considered a typical lower bound for lossy image and video compression [48, 16]. The threshold of 20 dB is considered the lower bound for lossy wireless transmission [44].

Evaluation Metrics. We define improvement as decrease in execution time, expressed as a percentage of the competitive protocol. We define the speedup of our approach as $Speedup = \frac{Speed\ of\ our\ protocol}{Speed\ of\ compared\ protocol} - 1$

Setup. We split the input videos into three groups: training, validation and test, with a share of 50%, 25%, and 25% of the videoset. The experiments are done on an x86 server with a six-core Intel(R) Xeon CPU, 16 GB RAM, and Ubuntu Linux kernel 4.4. We used FFmpeg libarary version 3.0 (compiled with gcc 5.4.0).

5.1 Performance and Quality Comparison for End-to-End Workflow

Figure 3 presents the results of the end-to-end workflow for the nine different video processing pipelines over all videos from the test set. Each plot presents the speedup relative to the exact pipeline for the following configurations (from left to right):

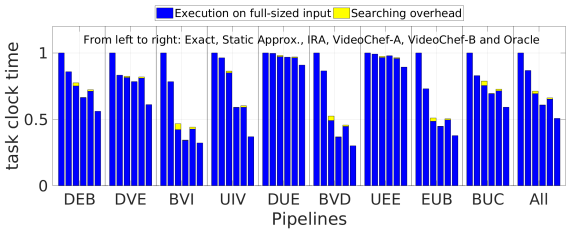
- Exact computation, with default parameters.
- Best static approximation, created by setting the AL that is just over the error threshold for *all* the frames in training videos.
- IRA extended with a simple searching policy that has a fixed interval of 10 frames. This number is chosen according to SAGE [35], which gives an analytic bound for a video processing setting.
- VIDEOCHEF version A – with I-frames detection.
- VIDEOCHEF version B – with scene change detection.
- Oracle version uses exhaustive search but does not incur search overhead. This sets the upper bound of the performance.

For both VIDEOCHEF versions, we used the CAD error model with 3dB margin, as the result of the analysis in Section 5.3.

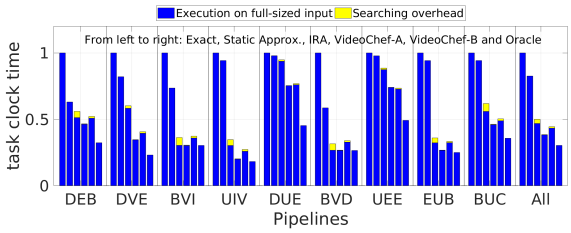
Performance for 30db Threshold. Figure 3(a) shows that VIDEOCHEF version A reduces the execution time by 39.1% over exact computation and is within 20% of the Oracle. It outperforms both static approximation and IRA, by respectively 29.9% and 14.6% in the aggregate. The advantage exists for all the video filter pipelines with the greatest savings relative to IRA being in Unsharp-Inflate-Vignette (UIV) pipeline. We are 39.2%, 36.8% and 29.5% better than exact computation, static approximation, and IRA, respectively. The search overhead for VIDEOCHEF (both versions A and B) is small – the yellow portions of the bars are almost not visible – and yet it finds more aggressive approximations than the competitive approaches (static or IRA) (the blue portions of the bars are shorter). The IRA approach, due to its assumption that the error in the canary output is identical to the error in the full output, cannot use aggressive ALs and thus cannot achieve the full speedup available through approximation. Within the two variants of VIDEOCHEF, scene change detector (version B) is slower than an I-frame lookup (version A).

Performance for 20db Threshold. We also evaluate on VIDEOCHEF on a different quality thesholds 20dB. Given a larger error budget, Figure 3 shows that VIDEOCHEF is able to achieve more performace gain over exact computation (1.6x speedup). We also outperform static approximation and IRA by 53.4% and 23.1% and within 26.6% from the Oracle results. Notice that the pipelines where we achieve the maximum performance gain over IRA changes from UIV to DVE.

Quality for 30db Threshold. Figure 4(a) shows that IRA and static approximation both achieve much higher quality than what the user specified (30 dB), an undesirable outcome here since this comes at the expense of higher execution time. VIDEOCHEF on the other hand

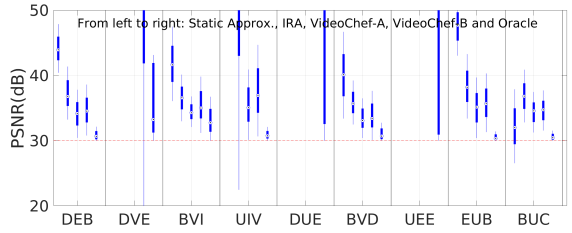


(a) Quality threshold = 30dB

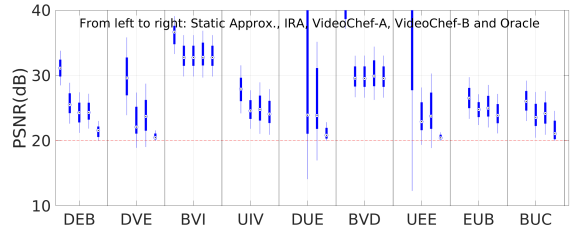


(b) Quality threshold = 20dB

Figure 3: Mean execution times over all frames of all videos. Geometric means of the speedups are on the right.



(a) Quality threshold = 30dB



(b) Quality threshold = 20dB

Figure 4: Quality of each frame across different video filter pipelines.

tracks the Oracle quality quite closely, which in turn meets the user requirement. It does however, drop below the threshold on some inputs, albeit by small amounts. This indicates that a future design feature should compensate for the tendency of VIDEOCHEF to sometime drop below the target video quality, say by adding a penalty function when the AL brings it close to the boundary. Further, a carefully designed margin in the searching algorithm can reduce the violation in quality requirement but still achieve speedup. The careful reader would have noticed that for some pipelines, some protocol results are missing here. This happens because no approximation is possible for some pipelines and there is no error introduced and hence, PSNR is not defined.

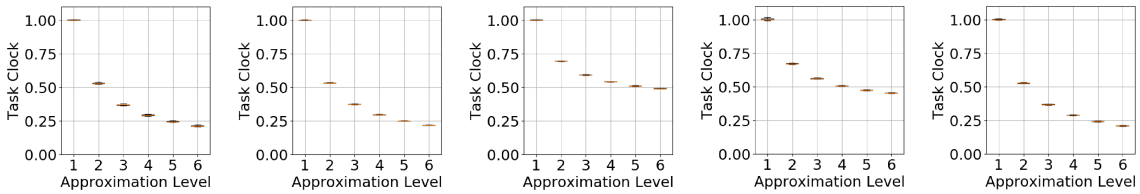
We also use the percentage of frames that violate the quality threshold to chracterize the robustness of each protocol. The violation rate of static approximation, IRA, VIDEOCHEF version A and B are 3.27%, 0.64%, 6.6% and 4.79%. Although the two versions of VIDEOCHEF have higher violation rates, they are still within a typical user acceptable threshold (5%). We consider the violation may due to two factors – (1) Inaccurate error prediction in the key frame. (2) The quality of non-key frames degrade and drop below the threshold before a fresh key frame is identified and a search triggered. According to our modeling in Sec 5.3, the violation due to the first factor is limited to at most 5%, while the second error may be inevitable as long as we do not search for every frame. Considering the trade-off between searching overhead and better error control, VIDEOCHEF is able to largely reduce the searching overhead and still maintain good quality.

Quality for 20db Threshold. Figure 4(b) shows the quality measurement of different protocols across all the pipelines. The mean violation rate averaged across all pipelines of static approximation, IRA, VIDEOCHEF version A and B are 0%, 0.23%, 7.18% and 3.93%. In the two quality threshold case, we see the advantage of scene change detection as an add-on in VIDEOCHEF version B to decrease the violation rate because it can accurately detect the frame which differs largely from the previous and trigger a required search for optimal approximation levels.

5.2 Speedup and Video Quality versus Approximation Levels

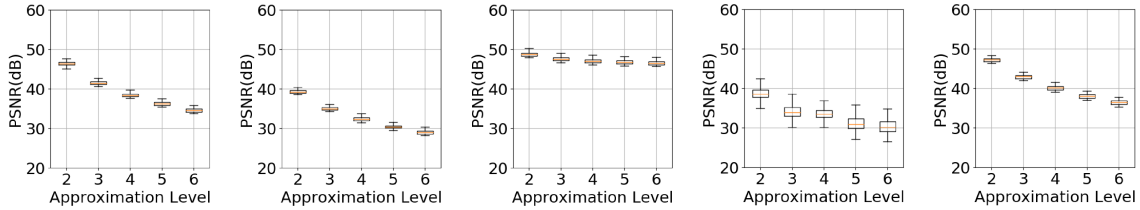
This experiment studies (1) how the execution time of each filter varies with the AL setting for that filter and (2) how the video quality varies with the AL setting. This result is dependent on the approximation technique but is independent of the VIDEOCHEF configuration used to decide on the AL. We show the results with all the videos in our dataset and 5 out of 10 representative filters in Figure 5 (number of executed instructions) and Figure 6 (video quality). When showing the result for a specific filter, we only execute on this filter and not the 3-stage pipeline. Here the results have higher variability due to the content-dependent effect. For the execution time, we normalize by the measure for exact computation.

Execution Time. Figure 5 shows that as the AL becomes higher, *i.e.*, the approximation becomes more aggressive, the execution time decreases. But the rate of decrease slows down as the AL becomes higher and the behaviors among the different filters in our evaluation are comparable. Note that this is a box plot, but there is little variation across the different videos and hence each AL gives



(a) Deflate filter (b) Emboss filter (c) Boxblur filter (d) Histeq filter (e) Vignette filter

Figure 5: The normalized execution time for each filter as the Approximation Level (AL) is varied, across all 106 videos in our dataset. The number of CPU cycles is normalized by the measure for exact computation. As the AL increases, the execution time decreases and this happens consistently across all videos and filters.



(a) Deflate filter (b) Emboss filter (c) Boxblur filter (d) Histeq filter (e) Vignette filter

Figure 6: The video quality for each filter as the Approximation Level (AL) is varied, across all 106 videos in our dataset. The effect depends on the video content and the filter being used.

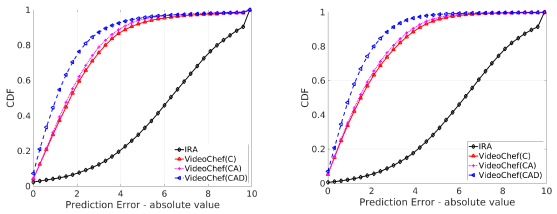
very tight result. This is expected because the amount of processing done in the filter, whether with exact or approximate computation, is *not* content dependent, but the effect of the approximation *is* content dependent.

Quality. Figure 6 shows the effect of AL on the video quality when the full frame is used. The quality degrades as the approximation gets more aggressive, but the nature of the decrease is not uniform across all the filters. Even within each filter, the effect on quality depends on the exact video frame, as implied by the vertical data spread for any given AL. We identify two forms of unpredictability of how AL correlates with video quality: with the content (which video frame is being approximated) and with the filter. Due to these two factors, we do not try to come up with a closed form curve for doing the prediction, rather, we do the actual computation with the canary input for a given AL setting, compute the PSNR, and then map it to the PSNR with the full input (Section 3.5). Contrast this to the execution time where we create a lookup table through training, which is content independent, and just look it up during the online search (Section 3.4.1). The variability due to video content in the PSNR plot validates our rationale for doing the approximation in a content-dependent manner. The rationale is shared with [24], but it sets our work apart from the approximation techniques that select the approximation configuration in a content-independent manner.

5.3 Evaluation of Error Mapping Models

In this experiment, we evaluate the quality of the various error mapping models in VIDEOCHEF. We trained the model on the training video set. Table 2 and Figure 3 present the performance of our model on the validation videos. Figure 7 shows that even a simple model C can greatly reduce the prediction error relative to IRA. Also, as we increase the level of knowledge, the model achieves higher prediction accuracy and model-CAD performs the best due to its good use of the feature extraction from the frames. We can see that with our CAD model, we can successfully control the error within 2dB in 80% of the cases and within 3dB in 90% of the cases. Given these results, we set up a 3dB margin when mapping from the canary error to the full error.

The results on the test videos (Section 5.1) show that the cases when we violate the quality requirement are within 10%.



(a) Validation set (b) Test set

Figure 7: Results of the error modeling in VIDEOCHEF mapping error in canary output to error in full output. The CAD model with characteristics of the frame performs best, though it is only slightly better than the CA models.

Table 2: F-1 measure of different error mapping models averaged over all pipelines. We regard IRA as a pass through error mapping.

Models	IRA	C	CA	CAD
30dB threshold	0.8650	0.9576	0.9594	0.9686
20dB threshold	0.8007	0.9679	0.9660	0.9759

Table 3: Results of the user studies with 16 videos processed using Oracle and VIDEOCHEF

Degree of difference	Percentage
No difference	58.59%
Little difference	34.77%
Large difference	6.64%
Total difference	0

5.4 User Perception Study

To evaluate if the protocols cause any perceptual difference, we conduct a small user study with 16 participants. Users were recruited by emailing students of certain ECE classes. We picked 16 videos, 2 from each YouTube content category, randomly picked from our dataset. We processed each video (a snippet of 20 seconds from each, as in the rest of the evaluation) using the Oracle approach and using VIDEOCHEF for pipeline DBE.

This pipeline was chosen because its result in the rest of the evaluation is representative and it produces videos which are still visually pleasing. In the experiment, we showed the two versions of each of the 16 videos concurrently, processed using the Oracle and VIDEOCHEF tools, without letting the participant know which window corresponded to which tool. All participants watched the videos independently. The participants were asked to rate the videos in four categories: Same, Little difference, Large difference, and Total difference. We gave guidance to the participants for the four categories as difference $\in$ [0%,5%),[5%,20%),[20%,50%), and $\geq$ 50%.

We show the results in Table 3. The percentage figure is the percentage of the total number of videos shown, which is 16 $\times$ 16 (number of videos $\times$ number of users). We conclude that 58.59% of the videos got no difference rating between the Oracle and the VIDEOCHEF processed videos, while 34.77% got a little difference rating. Although 6.64% of videos got large difference rating, none of the videos got total difference rating. This validates that qualitatively human perception is not seeing significant difference in video quality due to approximate processing using VIDEOCHEF.

6 Related Work

Approximate Tradeoffs in Computations and Data. Researchers presented various techniques for changing computations at the system level to trade accuracy for performance, *e.g.*, in hardware [30, 47, 12, 11, 8], runtime sytems [3, 18], and compilers [27, 41, 2, 38, 6]. A key challenge of approximate computing is finding good tradeoffs between accuracy and performance. For this, researchers have looked at both off-line autotuning [27, 41, 26, 37] and on-line dynamic adaptation [3, 18, 36, 22, 17]. In image processing, various techniques exist for synthesizing approximate filter versions, e.g., using genetic programming [45, 40, 13]. Recently, Lou et al. [25] present "image perforation", an adaptive verision of loop perforation tailored for individual image filters. Researchers also proposed storing multimedia data in approximate memories, including standard [38, 32], solid-state [39], and multi-level cell memories specialized for video encodings [19]. We consider such storage approaches complementary to our computation-based technique for video encoding.

Input-Aware Approximation. Several techniques provide input-aware approximations to monitor output quality and control the aggressiveness of the approximation during execution. Green [3] was an early approach that applied dynamic quality monitoring to adjust the level of approximation, based on a user-defined quality function. More recently, input-aware approximation identifies classes of similar inputs and applies different approximations for each input class [9, 43]. Opprox [29] learns the control-flow of the input-optimized program and then selects in which phase to approximate as well as how much to approximate. In contrast to our work, all these approaches use off-line models for prediction of input quality and do not craft the smaller inputs at runtime. Ringenburg *et. al.* [33] proposed online monitoring mechanisms, where a random subset of approximate outputs is compared with a precise output on a sampling basis, or the output of the current execution is predicted from past executions with similar inputs. Raha *et al.* [31] present a precise analysis of accuracy for a commonly used reduce-and-rank computational pattern. Rumba [22] and Topaz [1] detect outliers in intermediate computation results. In contrast to IRA [24] and our VIDEOCHEF, these approaches do not use canary inputs to guide the optimization and monitoring and therefore grapple with the overhead issue.

7 Conclusion

Fast and resource efficient processing of videos is required in many scenarios. We built a resource efficient and input-aware approximate video processing pipeline called VIDEOCHEF. VIDEOCHEF controls the approximation in each frame (using the properties of the frame) to meet the user's accuracy requirement. In particular, VIDEOCHEF uses a canary-input based approach for fast searching, as proposed in prior work, but overcomes some fundamental challenges by innovating a machine-learning based accurate error estimation technique and an input-aware search technique that finds best approximation settings. We show that VIDEOCHEF can provide significant speedup in 9 different video processing pipelines while satisfying user's quality requirements.

References

[1] Sara Achour and Martin C Rinard. Approximate computation with outlier detection in topaz. In *OOPSLA*, 2015.

[2] Jason Ansel, Yee Lok Wong, Cy Chan, Marek Olszewski, Alan Edelman, and Saman Amarasinghe. Language and compiler support for auto-tuning variable-accuracy algorithms. In *CGO*, 2011.

[3] Woongki Baek and Trishul M Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *PLDI*, 2010.

[4] Linchao Bao, Qingxiong Yang, and Hailin Jin. Fast edge-preserving patchmatch for large displacement optical flow. In *IEEE CVPR*, pages 3534–3541, 2014.

[5] Mark Buckler, Suren Jayasuriya, and Adrian Sampson. Reconfiguring the imaging pipeline for computer vision. In *The IEEE International Conference on Computer Vision (ICCV)*, 2017.

[6] Michael Carbin, Sasa Misailovic, and Martin C Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In *OOPSLA*, 2013.

[7] Swarat Chaudhuri, Sumit Gulwani, Roberto Lublinerman, and Sara Navidpour. Proving programs robust. In *FSE*, 2011.

[8] Vinay K Chippa, Debabrata Mohapatra, Anand Raghunathan, Kaushik Roy, and Srimat T Chakradhar. Scalable effort hardware design: exploiting algorithmic resilience for energy efficiency. In *DAC*, 2010.

[9] Yufei Ding, Jason Ansel, Kalyan Veeramachaneni, Xipeng Shen, Una-May O'Reilly, and Saman Amarasinghe. Autotuning algorithmic choice for input sensitivity. In *PLDI*, 2015.

[10] Edward R Dougherty and Roberto A Lotufo. *Hands-on morphological image processing*, volume 59. SPIE press, 2003.

[11] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Architecture support for disciplined approximate programming. In *ASPLOS*, 2012.

[12] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural acceleration for general-purpose approximate programs. In *MICRO*. IEEE Computer Society, 2012.

[13] Zeev Farbman, Raanan Fattal, and Dani Lischinski. Convolution pyramids. *ACM Trans. Graph.*, 30(6):175–1, 2011.

[14] Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *NSDI*, pages 363–376, 2017.

[15] Íngo Goiri, Ricardo Bianchini, Santosh Nagarakatte, and Thu D Nguyen. Approxhadoop: Bringing approximations to mapreduce frameworks. In *ASPLOS*, 2015.

[16] Raouf Hamzaoui and Dietmar Saupe. *Fractal image compression - in "Document and Image Compression"*. CRC Press, 2006.

[17] Henry Hoffmann. Jouleguard: energy guarantees for approximate applications. In *SOSP*, 2015.

[18] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. In *ASPLOS*, 2011.

[19] Djordje Jevdjic, Karin Strauss, Luis Ceze, and Henrique S Malvar. Approximate storage of compressed and encrypted videos. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 361–373. ACM, 2017.

[20] Haitao Jiang, Abdelsalam (Sumi) Helal, Ahmed K. Elmagarmid, and Anupam Joshi. Scene change detection techniques for video database systems. *Multimedia Systems*, 6(3):186–195, May 1998.

[21] Ben Juurlink, Mauricio Alvarez-Mesa, Chi Ching Chi, Arnaldo Azevedo, Cor Meenderinck, and Alex Ramirez. Understanding the application: An overview of the h. 264 standard. *Scalable Parallel Programming Applied to H. 264/AVC Decoding*, pages 5–15, 2012.

[22] Daya S Khudia, Babak Zamirai, Mehrzad Samadi, and Scott Mahlke. Rumba: an online quality management system for approximate computing. In *ISCA*, 2015.

[23] James Ko. Libvideo: A lightweight .net library to download youtube videos. https://github.com/jamesqo/libvideo, 2016.

[24] Michael A Laurenzano, Parker Hill, Mehrzad Samadi, Scott Mahlke, Jason Mars, and Lingjia Tang. Input responsiveness: using canary inputs to dynamically steer approximation. In *PLDI*. ACM, 2016.

[25] Liming Lou, Paul Nguyen, Jason Lawrence, and Connelly Barnes. Image perforation: Automatically accelerating image pipelines by intelligently skipping samples. *ACM Transactions on Graphics (TOG)*, 35(5):153, 2016.

[26] Jiayuan Meng, Srimat Chakradhar, and Anand Raghunathan. Best-effort parallel execution framework for recognition and mining applications. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12. IEEE, 2009.

[27] Sasa Misailovic, Stelios Sidiroglou, Henry Hoffmann, and Martin Rinard. Quality of service profiling. In *ICSE*, 2010.

[28] Subrata Mitra, Manish K Gupta, Sasa Misailovic, and Saurabh Bagchi. Phase-aware optimization in approximate computing. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization (CGO)*, pages 185–196. IEEE Press, 2017.

[29] Subrata Mitra, Manish K. Gupta, Sasa Misailovic, and Saurabh Bagchi. Phase-aware optimization in approximate computing. In *CGO*, 2017.

[30] Krishna V Palem. Energy aware computing through probabilistic switching: A study of limits. *IEEE Transactions on Computers*, 54(9):1123–1137, 2005.

[31] Arnab Raha, Swagath Venkataramani, Vijay Raghunathan, and Anand Raghunathan. Quality configurable reduce-and-rank for energy efficient approximate computing. In *DATE*, 2015.

[32] Ashish Ranjan, Arnab Raha, Swagath Venkataramani, Kaushik Roy, and Anand Raghunathan. Aslan: Synthesis of approximate sequential circuits. In *DATE*, 2014.

[33] Michael Ringenburg, Adrian Sampson, Isaac Ackerman, Luis Ceze, and Dan Grossman. Monitoring and debugging the quality of results in approximate programs. In *ASPLOS*, 2015.

[34] Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. Paraprox: Pattern-based approximation for data parallel applications. In *ASPLOS*, 2014.

[35] Mehrzad Samadi, Janghaeng Lee, D Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. Sage: Self-tuning approximation for graphics engines. In *MICRO*, 2013.

[36] Mehrzad Samadi, Janghaeng Lee, D. Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. Sage: Self-tuning approximation for graphics engines. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 13–24, 2013.

[37] Adrian Sampson, André Baixo, Benjamin Ransford, Thierry Moreau, Joshua Yip, Luis Ceze, and Mark Oskin. Accept: A programmer-guided compiler framework for practical approximate computing. *University of Washington Technical Report UW-CSE-15-01*, 2015.

[38] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: Approximate data types for safe and general low-power computation. In *PLDI*, 2011.

[39] Adrian Sampson, Jacob Nelson, Karin Strauss, and Luis Ceze. Approximate storage in solid-state memories. *ACM TOCS*, 2014.

[40] KC Sharman, AIE Alcazar, and Y Li. Evolving signal processing algorithms by genetic programming. In *Genetic Algorithms in Engineering Systems: Innovations and Applications, 1995. GALESIA. First International Conference on (Conf. Publ. No. 414)*, pages 473–480. IET, 1995.

[41] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *FSE*, 2011.

[42] Xin Sui, Andrew Lenharth, Donald S Fussell, and Keshav Pingali. Proactive control of approximate programs. In *ASPLOS*, 2016.

[43] Xin Sui, Andrew Lenharth, Donald S. Fussell, and Keshav Pingali. Proactive control of approximate programs. In *ASPLOS*, 2016.

[44] Nikolaos Thomos, Nikolaos V Boulgouris, and Michael G Strintzis. Optimized transmission of jpeg2000 streams over wireless channels. *IEEE Transactions on image processing*, 15(1):54–67, 2006.

[45] Kazuyoshi Uesaka and Masayuki Kawamata. Evolutionary synthesis of digital filter structures using

genetic programming. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 50(12):977–983, 2003.

[46] European Union. General data protection regulation. http://www.eugdpr.org/, 2017.

[47] Swagath Venkataramani, Vinay K Chippa, Srimat T Chakradhar, Kaushik Roy, and Anand Raghunathan. Quality programmable vector processors for approximate computing. In *MICRO*, 2013.

[48] Stephen T. Welstead. *Fractal and Wavelet Image Compression Techniques*. Society of Photo-Optical Instrumentation Engineers (SPIE), Bellingham, WA, USA, 1st edition, 1999.

[49] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J Freedman. Live video analytics at scale with approximation and delay-tolerance. In *NSDI*, pages 377–392, 2017.

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