CPU Microarchitectural Performance Characterization of Cloud Video Transcoding

Yuhan Chen Jingyuan Zhu Tanvir Ahmed Khan Baris Kasikci
University of Michigan
{chenyh, jingyz, takh, barisk}@umich.edu

***Abstract*—Video streaming accounts for more than 75% of all Internet traffic. Videos streamed to end-users are encoded to reduce their size in order to efficiently use the Internet traffic, and are decoded when played at end-users' devices. Videos have to be transcoded—*i.e.*, where one encoding format is converted to another—to fit users' different needs of resolution, framerate and encoding format. Global streaming service providers (*e.g.*, YouTube, Netflix, and Facebook) employ a large number of transcoding operations. Optimizing the performance of transcoding to provide speedup of a few percent can save millions of dollars in computational and energy costs. While prior works identified microarchitectural characteristics of the transcoding operation for different classes of videos, other parameters of video transcoding and their impact on CPU performance has yet to be studied.**

In this work, we investigate the microarchitectural performance of video transcoding with all videos from *vbench*, a publicly available cloud video benchmark suite. We profile the leading multimedia transcoding software, FFmpeg with all of its major configurable parameters across videos with different complexity (*e.g.*, videos with high motion and frequent scene transition are more complex). Based on our profiling results, we find key bottlenecks in instruction cache, data cache, and branch prediction unit for video transcoding workloads. Moreover, we observe that these bottlenecks vary widely in response to variation in transcoding parameters.

We leverage several state-of-the-art compiler approaches to mitigate performance bottlenecks of video transcoding operations. We apply AutoFDO, a feedback-directed optimization (FDO) tool to improve instruction cache and branch prediction performance. To improve data cache performance, we leverage Graphite, a polyhedral optimizer. Across all videos, AutoFDO and Graphite provide average speedups of 4.66% and 4.42% respectively. We also set up simulation settings with different microarchitecture configurations, and explore the potential improvement using a smart scheduler that assigns transcoding tasks to the best-fit configuration based on transcoding parameter values. The smart scheduler performs 3.72% better than the random scheduler and matches the performance of the best scheduler 75% of the time.

***Index Terms*—Video transcoding, workload characterization**

I. Introduction

Video streaming services are becoming increasingly popular, taking up a considerable portion of Internet traffic today. According to the Cisco Visual Networking Index report [4], video streaming took up 75% of the Internet traffic in 2017 and will take up 82% of the Internet traffic in 2022. Apart from video streaming, online gaming that also uses video traffic is rising rapidly, expected to grow 15 times by 2022. Figure 1 shows the Internet traffic from 2017 to 2022.

Video streaming service providers (*e.g.*, YouTube, Netflix, and Facebook) transfer (upload and download) only encoded videos to reduce video size and therefore corresponding Internet traffic. In most use cases, the uploaded video format differs from the distributed video format as the video distribution must support a wide variation in network bandwidth, screen resolution, and user preferences [34]. Consequently, streaming service providers apply a large number of video transcoding—the process of decoding an encoded video into raw frames and re-encoding those frames in a different encoding format [43]—operations. Therefore, performance optimization of video transcoding workloads has the potential of saving millions of dollars in computational and energy costs.

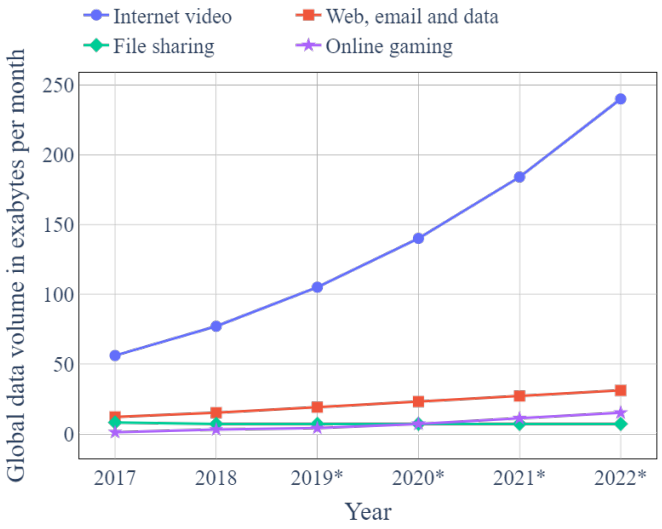


Fig. 1: Data volume of global consumer internet traffic from 2017 to 2022, by subsegment (in exabytes per month) [4]. The trend shows a rapid growth of video traffic both in absolute and relative terms.

The performance implications of video transcoding have inspired a rich set of prior works. Existing works have compared the performance of different transcoding algorithms [23], [35] as well as variation in transcoding performance for different videos [34]. While these works fill some gaps in understanding video transcoding workloads, there are several open questions in CPU microarchitectural bottleneck identification for these workloads. In this work, we aim to answer these questions by studying the microarchitectural characteristics

of video transcoding operations in response to variation in different transcoding parameters and inputs.

For performance characterization of video transcoding workloads, we utilize a wide range of CPU hardware performance counters using Intel VTune [8] and Linux perf [10]. Specifically, we leverage the Top-down Microarchitecture Analysis Method [45] to identify bottlenecks in the CPU microarchitecture for different video transcoding operations. Based on this methodology, we investigate the performance of the leading video transcoding software, FFmpeg. FFmpeg offers a large number of options to balance between transcoding speed, transcoded video quality, and transcoded file size. We examine how different values of these parameters affect microarchitectural performance issues for video transcoding workloads.

The intrinsic complexity of videos also affects transcoding performance. Videos with high motion and frequent scene transitions are of higher complexity, and they require longer transcoding time and a larger file size under the same quality constraint. *vbench* [34] is a benchmark developed for cloud video services. It uses clustering techniques to select 15 videos to cover a significant cross-section of a corpus of millions of videos. We use *vbench* to show how different video complexity affects the performance of video transcoding.

In this work, we make the following contributions:

- We identify key performance bottlenecks in CPU microarchitecture for a wide range of video transcoding workloads. Specifically, we observe that instruction cache, data cache, and branch prediction units suffer from frequent inefficiency for video transcoding operations. Moreover, we find that microarchitectural performance issues change rapidly due to variations in transcoding options and video complexity.
- We leverage state-of-the-art profile-guided optimization technique (AutoFDO [21]) to improve instruction cache and branch prediction performance of video transcoding workloads. We also apply a polyhedral optimizer (Graphite [37]) to improve data cache performance of video transcoding operations. AutoFDO provides 4.66% average speedup while Graphite provides 4.42% average speedup across workloads.
- We design a scheduler that assigns different video transcoding tasks to processors with different microarchitecture configurations based on transcoding parameters and inputs. In a simple case study, our designed scheduler performs 3.72% better than the random scheduler and matches the performance of the best scheduler 75% of the time.

The rest of the paper is organized as follows: we provide the necessary background of video transcoding in §II. We describe the experimental methodology of our paper in §III. §IV reports our experimental evaluation results. We briefly summarize the related works in §V. Finally, we conclude in §VI.

II. Background

Streaming videos have several important properties. A series of raw image frames constitute a video and the number of pixels in each frame is defined as video resolution. For example, a full high-definition (Full-HD) video contains 1920×1080 pixels per frame and is also known as a 1080p video. The number of frames for each second of video is defined as frame rate and expressed in frames per second or *FPS*. Streaming service providers support videos of different frame rates (24-60 *FPS*). Without any compression, a single-second standard video (with 1080p resolution and 30 *FPS* frame rate) requires 178 MB of space [14]. To reduce this high storage and network transmission cost, videos are encoded in several standard formats.

A. Video Transcoding

Video transcoding is the process of converting one encoding format to another and it is necessary because streaming service end-users have different requirements in terms of video resolution, frame rate, and encoding format based on their device capability and network condition. Today, more than 500 hours of videos are uploaded to YouTube every minute [7]. Since each uploaded videos must be transcoded at least once [34], streaming service providers perform an extremely large number of video transcoding operations. Moreover, the cost of performing video transcoding is expensive. For instance, Amazon Elastic Transcoder charges 0.03$ to transcode a single-minute video clip [2]. In this rate, transcoding 500 hours of videos will require around 1800$.

Video transcoding is performed in two stages: (1) an encoded video is decoded into raw frames, (2) these frames are encoded again to a different format. The decoding stage is deterministic and hence relatively straight-forward. On the other hand, the encoding stage is much more complex as it models the video compression problem as a heuristic-driven search space exploration problem. Moreover, the encoding stage has two primary components: (1) *Intra-frame encoding* compresses pixels within a single frame by eliminating spatial redundancy and (2) *Inter-frame encoding* compresses pixels across different frames by eliminating temporal redundancy. The intra-frame encoding divides a frame into several macroblocks [27]. On the other hand, the inter-frame encoding categorizes each frame as I (Intra-coded), B (Bidirectional predicted), or P(Predicted) picture frame [13].

FFmpeg is the leading video transcoding framework that can perform a wide range of operations (*e.g.*, decoding, encoding, transcoding, filtering, multiplexing, etc.) for different video encoding formats [1]. Since FFmpeg is the most widely used video transcoding software, we specifically focus on FFmpeg workloads. FFmpeg is typically compiled with x264, an open-source library developed by VideoLAN that implements state-of-the-art video encoding algorithms [16].

B. x264 Encoder

x264 is the state-of-the-art video encoder [16]. x264 achieves high performance with its rate control, motion estimation, macroblock mode decision, and quantization algorithms. The details of the algorithms are out of the scope of this work, but we describe what each of them does as we focus on how different algorithm parameters affect the transcoding performance.

1) Rate Control: Rate control is the mechanism to impose a constraint on bitrate or quality. It can be performed at three different granularities: at a coarse-grained level for a

group of pictures (GOP), for a single picture, and at a fine-grained level for macroblocks. There are mainly six rate control modes: constant QP (CQP) controls the amount of quantization; average bitrate (ABR) tries to achieve the target average bitrate; 2-pass average bitrate (2-Pass ABR) is similar to ABR except it runs twice as the first pass provides a better estimation for the second pass encoding; constant bitrate (CBR) imposes a constant bitrate; constant rate factor (CRF) controls the quality rather than the bitrate, and constrained encoding (VBV) constrains the bitrate to a certain maximum. Among the six modes, only CBR is applied at the granularity of a macroblock. Other modes are applied at the granularity of the picture.

2) Motion Estimation: Motion estimation is the most complex and time-consuming component of x264 encoding process. It detects the motion of objects (*e.g.*, translation, rotation, and tilting), encodes only the motion information, and thus saves space by not storing the entire frame. x264 provides four integer-pixel motion estimation methods: diamond (dia), hexagon (hex), uneven multi-hexagon (umh), and exhaustive (esa). Each mode represents a different search pattern, each more complex and time-consuming than the previous, but generates better motion estimation.

3) Macroblock Mode Decision: When encoding, each frame is partitioned into 16×16 macroblocks, and the macroblocks can be further partitioned into smaller blocks.An I-frame can only have I-macroblocks because it must not depend on other frames to decode, a P-frame can have both I-macroblocks and P-macroblocks, a B-frame can have I-macroblocks, P-macroblocks, and B-macroblocks.

4) Quantization: After motion estimation and macroblock mode decision, the residue between the original frame and prediction frame is computed. The x264 encoder uses trellis quantization [36], [41] to improve the storage efficiency of the residue. Users can select any one from three different levels of trellis quantization provided by the x264 encoder.

C. CPU vs. GPU

Videos can be transcoded in both CPUs and GPUs [5]. Typically, GPUs are faster than CPUs in terms of video transcoding time. However, GPUs perform worse than CPUs in terms of video compression ratio and video quality. Hence, GPUs are leveraged to transcode only live-streamed videos where transcoding speed matters more than the transcoded video size or quality. Moreover, video transcoding in GPUs is relatively new and supports only a subset of video formats [34]. In practice, GPUs are used only as a hardware accelerator instead of the primary transcoder [17]. Therefore in this work, we focus on video transcoding in CPUs.

D. Video Selection

Randomly-selected videos could lead to biased and unrepresentative profiling results. In this work, we use videos from *vbench* benchmark suite [34]. The videos from *vbench* benchmark suite are representative of cloud transcoding workloads. *vbench* uses clustering techniques to select 15 videos of 5 seconds each from a corpus of millions of videos [34], and therefore is diverse and representative of real videos. We study microarchitectural characteristics of the video transcoding operation for all *vbench* videos. We also use a video called Big_Buck_Bunny [3], widely studied in prior works [29], [33]. We list the detailed information of each studied videos in Table I. *vbench* also introduces a new video property, entropy to represent the complexity of a video. This property specifies the number of bits required to encode a video with the visually lossless quality [34]. A higher entropy suggests the video is more complex, for example, involves more motion, or frequent scene transition, and thus requires more computing resources and higher bitrate.

TABLE I: *vbench* videos info

Full Name	Short Name	Resolution	FPS	Entropy
desktop_1280x720_30.mkv	desktop	720p	30	0.2
presentation_1920x1080_25.mkv	presentation	1080p	25	0.2
bike_1280x720_29.mkv	bike	720p	29	0.9
funny_1920x1080_30.mkv	funny	1080p	30	2.5
cricket_1280x720_30.mkv	cricket	720p	30	3.4
house_1920x1080_30.mkv	house	1080p	30	3.6
game1_1920x1080_60.mkv	game1	1080p	60	4.6
game2_1280x720_30.mkv	game2	720p	30	4.9
girl_1280x720_30.mkv	girl	720p	30	5.9
chicken_3840x2160_30.mkv	chicken	2160p	30	5.9
game3_1280x720_59.mkv	game3	720p	59	6.1
cat_854x480_29.mkv	cat	480p	29	6.8
holi_854x480_30.mkv	holi	480p	30	7
landscape_1920x1080_29.mkv	landscape	1080p	29	7.2
hall_1920x1080_29.mkv	hall	1080p	29	7.7

III. Methodology

Hardware platforms. We use a 4-core 3.5GHz Intel Xeon E3 CPU (NUMA with 1 socket). The memory hierarchy of the machine consists of 64KB of private L1-cache (32KB private instruction and 32KB private data), 256KB of private L2 cache, 8MB of shared L3 cache, and 16GB of RAM.

Software platforms. All experiments are conducted in Ubuntu 16.04 (Linux kernel version 4.15.0) using GCC version 5.5.0, ffmpeg version N-82144-g940b890, and x264 version 148-r2762-90a61ec.

A. Transcoding metrics and parameters.

Video transcoding workloads maintain a unique trade-off among three key performance metrics: (1) transcoding speed (measured by transcoding time in seconds), (2) transcoded video quality (measured by peak signal to noise ratio [PSNR] in decibels [dB]), and (3) transcoded video file size (measured by bitrate in Kbps or Mbps). FFmpeg in combination with x264 provides many encoding options to balance among these three performance metrics. Among all such options, the most important parameters are *crf* and *refs* [12], [15] and therefore, we investigate the microarchitectural performance implications of video transcoding in response to variation in these two parameters. Figure 2 shows how these parameters (*crf* and *refs*) affect key transcoding metrics (speed, quality, and size).

crf actively controls the transcoded video quality. An increase in *crf* value results in video quality degradation after encoding. In x264 encoding, *crf* can be varied from 0 to 51.

Videos encoded with *crf* 0 are lossless while videos encoded with *crf* 51 are of worst quality. x264 uses 23 as the default value for *crf*. *crf* also has a passive impact on transcoding speed and transcoded file size. An increase in *crf* value results in faster transcoding time and smaller transcoded file size.

On the other hand, *refs* directly controls the transcoded video file size. *refs* (Reference frame number) specifies how many reference frames will be used during inter-frame encoding in addition to the frame immediately prior to the current frame [9]. In x264 encoding, *refs* can be varied from 1 to 16. An increase in *refs* value expands the encoding search space, improves the compression possibility, and hence reduces transcoded video file size. However, an increase in *refs* value also slows down the transcoding process due to larger search space exploration. *refs* has no impact on transcoded video quality.

In addition to *crf* and *refs*, we also study the performance impact of different x264 presets (a combination of standard values for all transcoding parameters) that vary other transcoding options including motion estimation, macroblock mode decision, quantization, and frame type decision.

B. Tools

1) VTune: The Intel VTune profiler [8] is a performance analysis tool that leverages a large number of hardware performance counters provided by Intel Performance Monitoring Unit (PMU) [11]. Specifically, VTune uses Top-down microarchitecture analysis method [45] to identify performance bottlenecks for CPU workloads. In Top-down methodology, performance issues are categorized into four major categories—retiring, bad speculation, front-end bound, and back-end bound—measured in the percentage of pipeline slots. A pipeline slot represents hardware resources needed to process one micro-operation (μOp). Ideally, the pipeline slots should be filled with instructions and successfully retire, but limited resources or bad speculations can lead to wasted pipeline slots.

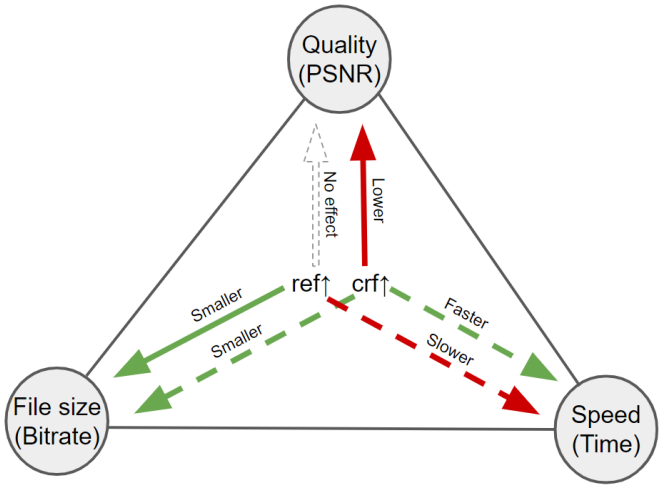


Fig. 2: Transcoding speed, video quality, and file size triangle. It shows the effects of increasing *crf* and *refs* on the three metrics. A green line denotes a positive impact, a red line represents a negative impact, a solid line denotes an active impact (purpose of changing the option), and a dotted line indicates a passive impact (side effect).

Front-end bound pipeline slots are unused due to issues like instruction cache misses and instruction decoder unavailability. On the other hand, back-end bound slots are unused because of problems including data cache misses (memory bound) and computational unit shortage (core bound). Bad speculation issues are mainly due to branch mispredictions. Finally retired slots denote properly utilized pipeline slots.

We leverage VTune to understand how different parameters and video workloads affect microarchitectural performance problems during transcoding. Particularly, we investigate how front-end bound, bad speculation, and back-end bound issues are affected by different transcoding settings. Moreover, we use VTune to determine the root cause of performance problems.

2) Linux perf: Linux perf [10] provides a simple command-line interface to profile CPU executions. We leverage perf mainly to reveal more fine-grained details such as L1, L2, L3, and branch misses per kilo instructions (MPKI).

3) AutoFDO: AutoFDO [21] is the state-of-the-art feedback-directed optimization (FDO) tool. AutoFDO captures the frequently-taken branches and optimizes their layout to reduce instruction cache misses and branch mispredictions.

4) Graphite: Graphite [37] is a polyhedral analysis and optimization tool for GCC. It uses the polyhedral model to optimize nested loops, where optimizations like loop tiling and loop fusion can be applied to enable better cache locality. We use graphite to reduce back-end stalls during the transcoding operation by improving L1, L2, and L3 cache hit rates.

5) Sniper: Sniper [20] is an open-source x86 simulator that can accurately simulate CPU executions with high speed. Moreover, Sniper allows modifying different microarchitecture parameters to study the performance impact of varying different processor configurations. We utilize Sniper to simulate our proposed scheduling algorithm with multiple μarch configurations.

C. Profiling Setup

1) Across crf & refs: We vary different transcoding parameters (*crf* from 1-51 and *refs* from 1-16) and investigate 816 different combinations for a single video and study the profiling results. We use VTune to capture the high-level profiling results grouped into four categories, and then use perf to get fine-grained results.

2) Across Presets: The x264 encoder provides 10 predefined setups (presets) to vary different transcoding parameters for different usage scenarios [6]. We list parameter values for these presets in Table II. All presets also specify a *crf* and *refs* number. Since, we investigate the performance impact of *crf* and *refs* separately, we use the default *crf* (23) and *refs* (3) values for different presets. We investigate the performance impact of different presets for a single video.

3) Across Videos: Finally, we study the performance of transcoding for a wide range of videos from the *vbench* benchmark suite with the parameters *crf* =23, *refs* =3, and x264 preset being medium.

TABLE II: Selection of the important options for different presets, adapted from [6]

Option	ultrafast	superfast	veryfast	faster	fast	medium	slow	slower	veryslow	placebo
aq-mode	0*	1	1	1	1	1	1	1	1	1
b-adapt	0*	1	1	1	1	1	1	2*	2*	2*
bframes	0*	3	3	3	3	3	3	3	8*	16*
deblock	[0:0]*	[1:0]	[1:0]	[1:0]	[1:0]	[1:0]	[1:0]	[1:0]	[1:0]	[1:0]
me	dia*	dia*	hex	hex	hex	hex	hex	umh*	umh*	tesa*
merange	16	16	16	16	16	16	16	16	24*	24*
partitions	none*	+i8x8,+i4x4*	-p4x4	-p4x4	-p4x4	-p4x4	-p4x4	all*	all*	all*
refs	1*	1*	1*	2*	2*	3	5*	8*	16*	16*
scenecut	0*	40	40	40	40	40	40	40	40	40
subme	0*	1*	2*	4*	6*	7	8*	9*	10*	11*
trellis	0*	0*	0*	1	1	1	2*	2*	2*	2*

* value differ from medium (default) preset

D. Optimization Setup

Implementing new optimizations for video transcoding is not the primary focus of this work. Instead, we study the impact of several optimizations to show the potential for improvement.

1) AutoFDO & Graphite: We optimize the video transcoding operation using AutoFDO to avoid instruction cache misses (grouped under front-end issues in Top-down methodology) and branch mispredictions (grouped under bad speculation issues in Top-down methodology). To apply AutoFDO, we use the FFmpeg program to transcode multiple videos and collect execution profiles using perf during transcoding. Then, we optimize FFmpeg by recompiling the program with the collected profile.

We optimize the video transcoding operation using Graphite to reduce data cache misses (grouped under back-end issues in Top-down methodology). Graphite is integrated into GCC and therefore can be directly used by enabling specific optimization flags (`-floop-interchange -ftree-loop-distribution -floop-block`) during compilation. We enable those optimization flags during the compilation of the FFmpeg program.

2) Smart Scheduler: Streaming service providers may have transcoding servers with different µarch configurations. Even without optimizing the algorithm or implementation, knowing how to intelligently assign tasks to the server that best fits the task can fully utilize the resources and save transcoding time. The profiling results can be used as a reference to schedule transcoding tasks to the fitting server.

We consider four transcoding tasks, each with different video, *crf*, *refs*, and preset combinations as shown in Table III. We also modify the baseline µarch configuration of the Sniper simulator (gainestown) to create 4 µarch configurations. Different µarch configurations are optimized to reduce different types of pipeline issues by varying different microarchitectural resources. The baseline and four modified configurations are described in Table IV. We use different strategies to assign tasks to different µarch configurations (servers), and use the Sniper simulator to measure the transcoding time.

TABLE III: Transcoding parameters used for Sniper simulation

Task#	Video	*crf*	*refs*	Preset
1	desktop	30	8	veryfast
2	holi	10	1	slow
3	presentation	35	6	veryfast
4	game2	15	2	medium

We evaluate three different schedulers. The random scheduler randomly assigns tasks among servers, so we use the average value of all four servers as its performance. The smart scheduler assigns tasks to the best-fit server, under the constraint that the four tasks must be assigned to different server (one-to-one constraint), preventing any server from over-utilizing or under-utilizing. Finally, the best scheduler assigns tasks to the best-fit server without the one-to-one constraint.

IV. Evaluation Results

A. Profiling

1) Across crf & refs: Figure 3 shows the heatmaps of 816 combinations where *crf* is varied from 1 to 51 and *refs* is varied from 1 to 16. The projections of each data point to the three axes represent video quality (PSNR), file size (bitrate), and transcoding speed (time) respectively, and the color represents either front-end, back-end, or bad speculation bound percentage in terms of pipeline slots. All three heatmaps are of the same shape, but represent different bounds. As shown in Figure 3, both increasing *crf* and *refs* reduce front-end and bad speculation bound slots, but increases back-end bound slots.

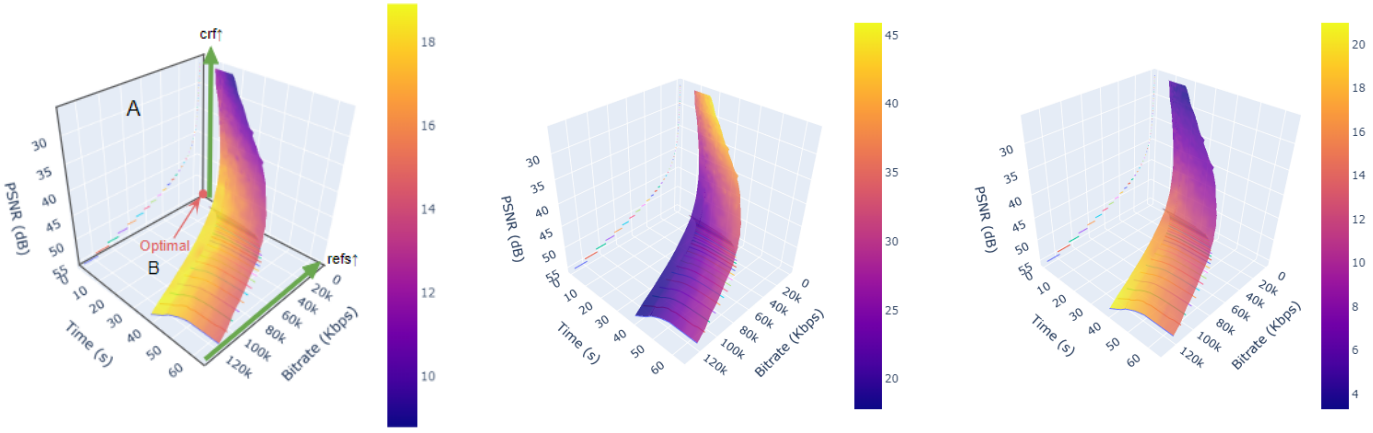
Figure 4 shows two projections into plane A and B from Figure 3. Projection A has 51 horizontal lines for 51 discrete PSNR values. Each line stands for one *crf* value as *crf* controls the video quality. With *crf* fixed, increasing *refs* can help saving file size. The length of each horizontal line shows the range of bitrate when increasing *refs* from 1 to 16. The longer the horizontal line is, the more we can benefit from increasing the *refs* value. With *crf* increasing, PSNR decreases, meaning the video quality deteriorates. Also with *crf* increasing, the length of the line decreases, denoting a diminishing return for increasing *refs*.

Projection B is the relation between time and *refs*, where increasing *refs* does not linearly decrease the file size. For each *crf*, there is an elbow point beyond which increasing *refs* has little or even no return. Moreover, increasing *crf* makes the line flatter, meaning high *crf* benefits less from increasing *refs*, which is in line with the conclusion from projection A.

TABLE IV: Different microarchitectural configurations for Sniper simulation. The baseline is the default configuration provided by Sniper, Gainestown. fe_op is optimized to reduce front-end stalls with larger L1i-cache and iTLB. be_op1 and be_op2 are optimized to reduce back-end stalls by increasing the capacity of data caches and other pipeline resources. bs_op is optimized to avoid bad speculation stalls by replacing the default pentium_m branch predictor with the Tage branch predictor.

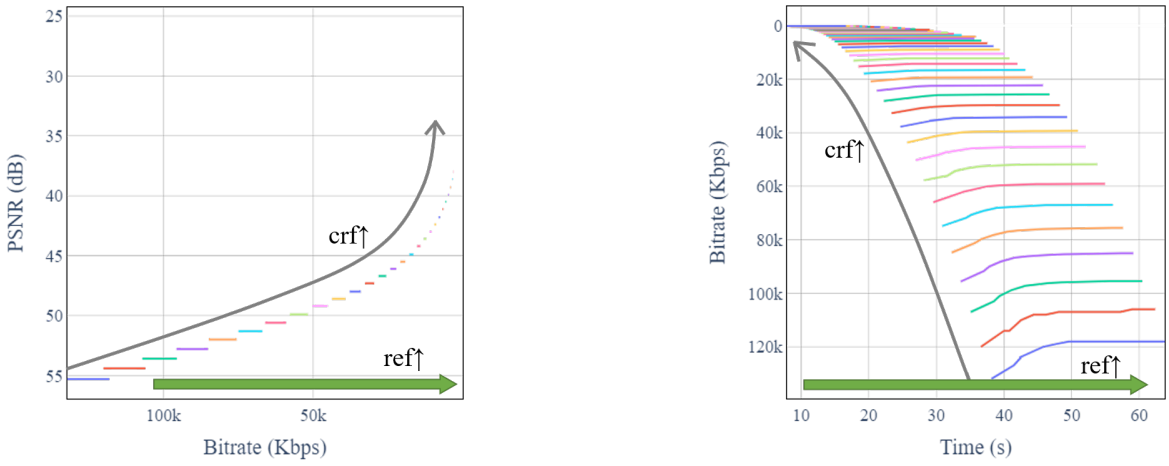
Config Name	L1d size	L1i size	L2 size	L3 size	L4 size	itlb size	ROB size	RS size	issue at dispatch	branch predictor
baseline	32K	32K	256K	8192K	none	128	128	36	No	Pentium_m
fe_op	-	64K	-	-	-	256	-	-	-	-
be_op1	64K	-	512K	4096K	16384K	-	-	-	-	-
be_op2	-	-	-	-	-	-	256	72	Yes	-
bs_op	-	-	-	-	-	-	-	-	-	Tage

- means same as baseline



(a) Front-end Bound (b) Back-end Bound (c) Bad speculation bound

Fig. 3: Heatmaps of front-end, back-end, and bad speculation bound pipeline slots (%)

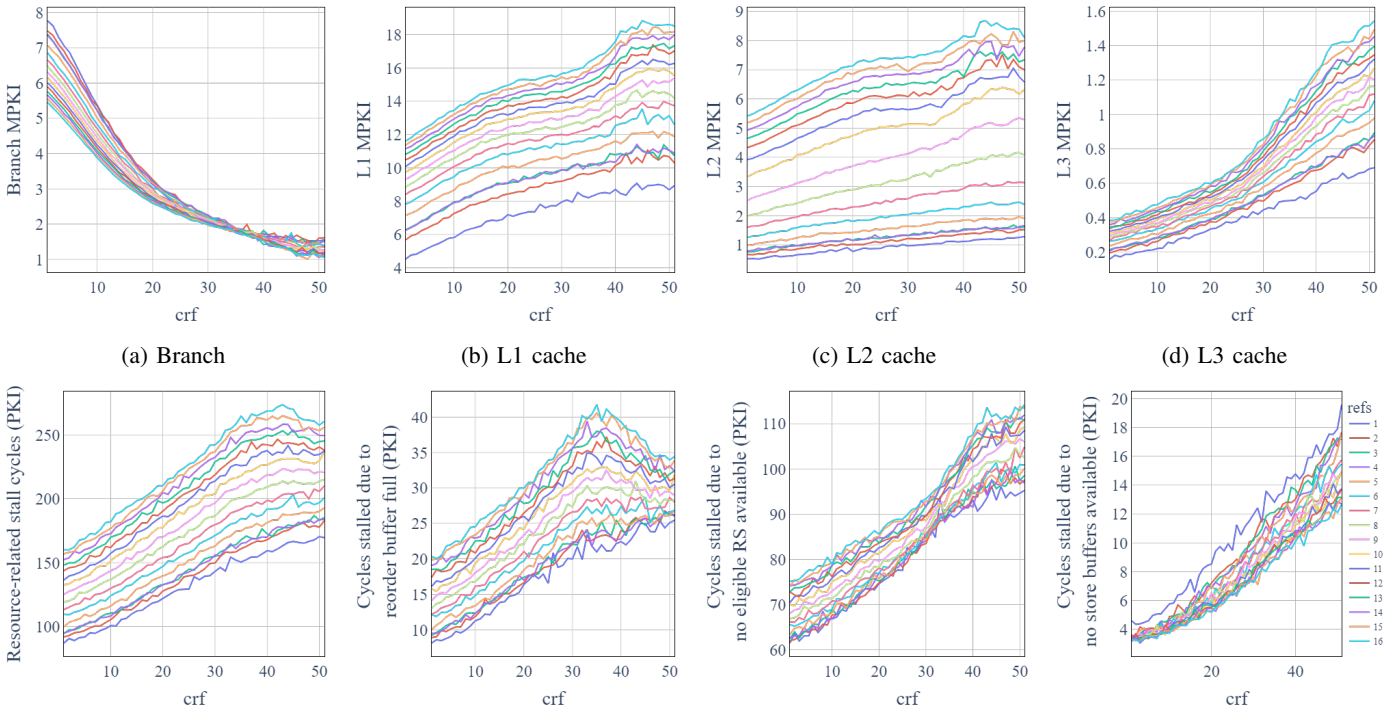


(a) Projection A (b) Projection B

Fig. 4: Projection A & projection B

The main takeaways from the three heatmaps and two projections are: low *crf* benefits more from increasing *refs*, and increasing *refs* has diminishing returns. The result is video dependent, and the elbow points can differ for different videos, but the trend shown is universally applicable.

Further analysis shows the front-end bound slots are mostly due to the inefficiency in micro-instruction translation engine (MITE), and decoded stream buffer (DSB), both related to decoding instructions (instruction to micro-op conversion). Front-end bound slots represent only a small fraction of overall pipeline slots and do not change significantly for different *crf*



(a) Branch (b) L1 cache (c) L2 cache (d) L3 cache

(e) Resource - Any (f) Resource - ROB (g) Resource - RS (h) Resource - SB

Fig. 5: Profiling results showing inefficiency in different microarchitecture resources for different values of transcoding parameters, *crf* and *refs*.

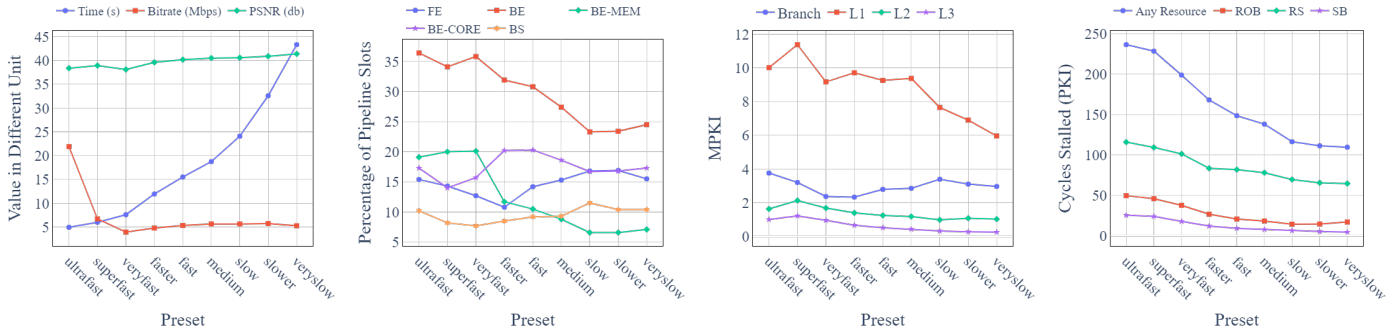
and *refs* combinations. Back-end issues are responsible for most of the wasted pipeline slots. The back-end bound issues can be further divided into memory bound and core bound problems. Memory bound slots means the pipeline is stalled because the required data is not available, and core bound means the pipeline is stalled because the hardware resources (functional units) needed to perform operations are not available. In our evaluation, Bad speculation bound slots are almost always due to branch mispredictions.

To further investigate these wasted pipeline slots, we evaluate the inefficiency in several microarchitectural resources. Specifically, we study the variation in eight hardware performance events in response to change in *crf* and *refs*. Figure 5 shows the results. Figure 5a shows that branch mispredictions per kilo instructions decreases when both *crf* and *refs* increase. Figure 5b, 5c, and 5d depicts misses per kilo instructions (MPKI) for L1, L2, and L3 data caches respectively. These cache misses are mainly responsible for the memory bound component within the back-end bound slots. Figure 5e, 5f, 5g, and 5h denotes inefficiency in pipeline execution units and constitute the core bound component within the back-end bound slot. All of these inefficiencies show a similar trend of deteriorating when either *crf* or *refs* increase. Here, store buffer (SB) efficiency is a notable exception as the number of stalls due to unavailable store buffer decreases when *refs* increases.

The trend shown in both memory bound and core bound issues can be explained using the roofline model [42], a performance model that correlates performance with operational intensity. The roofline model defines operational intensity as how much computation is performed for each byte of DRAM traffic. For low operational intensity, CPU performs little arithmetic operation on each piece of data, and the workload is bound by memory. As operational intensity increases, the utilization of CPU and the overall performance increase. When the operational intensity is high enough to occupy all CPU resources, the workload becomes compute bound.

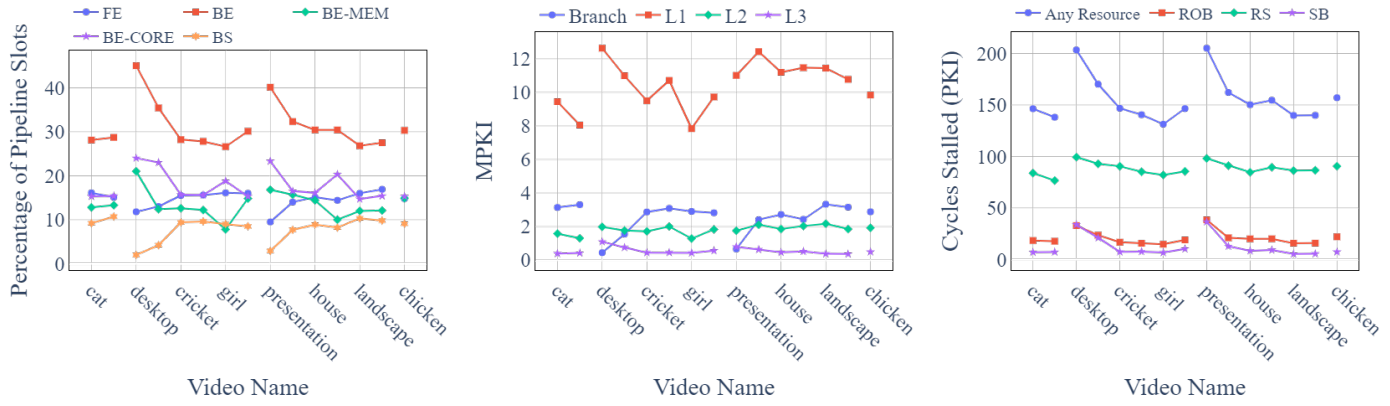
Increasing *crf* relaxes the quality constraint and requires less computation for the same amount of data transfer, thus causes a lower operational intensity. On the other hand, increasing *refs* increases both total number of executed instructions and memory traffic, but more on memory traffic, thus lowers the operational intensity as well. For lower operational intensity, processors have limited number of computations to hide the memory latency which results in higher memory bound stalls.

The roofline model can also explain the lower amount of front-end bound slots. As the CPU is waiting for memory traffic, it exhausts the non-arithmetic resources (*e.g.*, Reorder buffer [ROB], reservation stations [RS], and store buffer [SB]) quickly. Consequently, the CPU stops fetching new instructions and therefore has lower amount of front-end bound stalls. Note that SB stalls show different trend compared to ROB and RS stalls with a change in *refs*. That is because, higher *refs* results in better video compression which requires less number of total store operations.



(a) Time, Bitrate, PSNR (b) FE, BE, BS (c) Branch, Cache MPKI (d) Resources

Fig. 6: Profiling results for different transcoding presets



(a) FE, BE, BS (b) Branch, Cache MPKI (c) Resources

Fig. 7: Profiling results for different videos

2) Across Presets: Figure 6a shows how transcoding time, bitrate, and PSNR change for different transcoding presets. Similarly, Figure 6b shows the percentage of front-end, back-end, and bad speculation bound slots (%) for different transcoding presets. From the fastest to the slowest preset, transcoding time increases as expected. As *crf* is fixed, PSNR has a minor increase while bitrate shows great improvement from ultrafast to superfast, and superfast to veryfast, and then shows diminishing or even no returns with any slower presets. The trend of transcoding time and bitrate suggests that without any strict time constraint, tuning up the preset to veryfast can trade a small increment in transcoding time for file size reduction. Figure 6c shows that the branch MPKI fluctuates with no clear direction. Data cache MPKI goes down while using a slower preset. The trend agrees with the Figure 6b, where only back-end issues have a clear trend of going down. This is mainly because the memory bound component decreases. Figure 6d shows stalls due to resource unavailability, which can also be explained with the roofline model [42]. A slower preset has a higher operational intensity, thus it is less likely to run into memory bound issues. Consequently, fewer instructions block ROB, RS and SB waiting for memory.

3) Across Videos: We now investigate the variation in microarchitectural characteristics while transcoding different videos. We first group videos based on different resolutions and then sort them based on different entropy [34]. The gaps in Figure 7 separate different resolution groups. As Figure 7a shows, with an increase in video entropy, front-end and bad speculation bound slots increase and back-end bound slots decrease. Figure 7b and 7c show the variation of branch misprediction, memory bound, and core bound slots for different videos. As branch mispredictions dominate the bad speculation issues for video transcoding workloads, branch MPKI and slots lost due to bad speculation follow similar trend. L1, L2, and L3 data cache MPKI follow the same trend as the memory bound slots. Similarly, stalls due to other pipeline resources follow the same trend as the core bound slots. The roofline model can also be applied here. Videos with higher entropy are more complex, and need higher operational intensity to encode under the same quality constraint, leading to lower back-end bound issues.

B. Optimization

1) AutoFDO & Graphite: We optimize FFmpeg with AutoFDO and Graphite to reduce front-end, bad speculation, back-end bound stalls while transcoding different videos. Figure 8

shows the results. AutoFDO provides an average speedup of 4.66%, with a maximum of 5.2%. On the other hand, Graphite provides an average improvement of 4.42%, with a maximum of 4.87%.

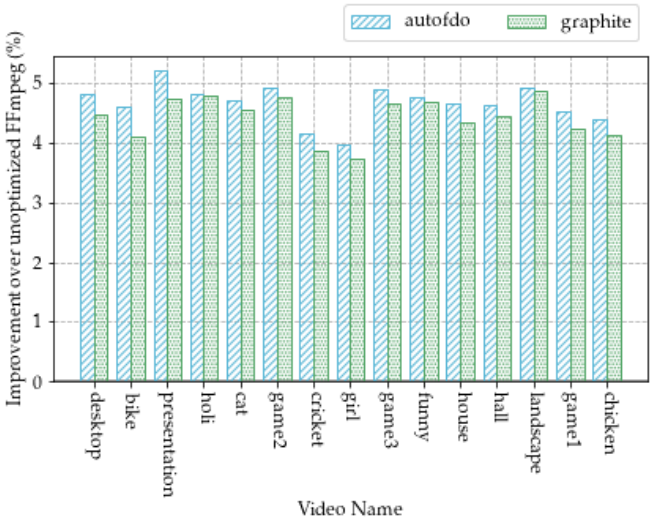


Fig. 8: Speedup provided by AutoFDO-optimized FFmpeg binary and Graphite-optimized FFmpeg binary. The number is the average of 32 different combinations of transcoding parameters (*crf*, *refs*, and presets).

2) Smart Scheduler: Figure 9 shows the speedup provided by three schedulers over the default configuration. As all four µarch configurations have better microarchitectural resources compared to the default baseline, all schedulers show performance gain. However, on average, our characterization-driven smart scheduler outperforms the random scheduler by 3.72%. Moreover, our smart scheduler provides the same schedule as the best-fit server in three out of four cases.

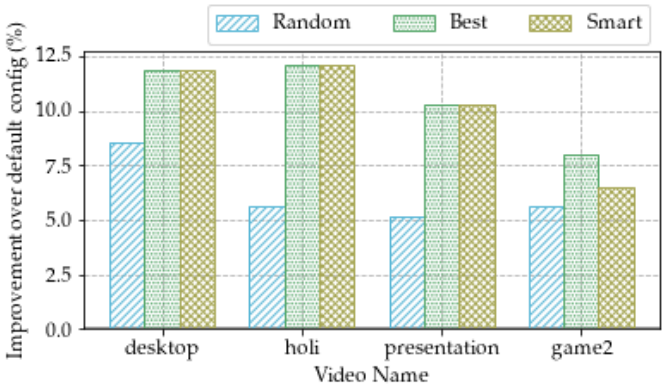


Fig. 9: Transcoding speedup over the baseline µarch configuration. The random scheduler uses the average improvement of four modified µarch configurations. The one-to-one constraint is imposed on the smart scheduler, but not on the best scheduler.

V. Related Work

The performance of video transcoding has a significant impact on computational and energy savings. Realizing this significance, a rich set of prior works has investigated the performance of video transcoding. We describe related works in four categories.

Performance profiling of video transcoding. Different prior works have investigated the performance of video transcoding from different perspectives. For example, COVT [40] measures the transcoding time and compression ratio for many transcoding presets and video types, and use the results for efficient resource allocation. Other works [22], [26] aims at predicting the power consumption of video transcoding workloads. In comparison, we focus on microarchitectural bottlenecks while transcoding various videos with different parameter values.

Algorithmic optimization. Many prior works have examined the algorithmic optimization of video transcoding operation. For example, parallel transcoding on the Cloud [19], [32] optimizes for parallelism and DCT transcoder [31] optimizes for fast DCT-domain transcoding. Sung et al. [39] propose a method to utilize the quadtree information from the decoding process to accelerate the encoding process. Zhang et al. [48] observe that the video background barely changes for certain types of videos and utilize this observation to achieve fast transcoding. In comparison, we notice the microarchitectural resource inefficiency during video transcoding and leverage state-of-the-art compiler optimizations to improve the hardware resource utilization.

System/architectural optimization. Several prior works have designed efficient systems for video transcoding. Specifically, [24], [49] optimizes the storage efficiency while transcoding videos in content delivery networks (CDNs). GPU-accelerated VTU for MEC [17] leverages GPUs to accelerate the transcoding operation. Cloud Transcoder [30] bridges the gap between internet videos and mobile devices by offloading the bulk of the transcoding operation from mobile devices to the cloud. Our characterization of video transcoding provides several insights on performance bottlenecks. These insights can be leveraged to design an efficient video transcoding system in the future as we show with our smart scheduler experiment.

Adaptive video streaming. Adaptive video streaming services tune video transcoding parameters to generate videos of different quality [18], [28], [38]. The values of these parameters are predicted based on the network condition [25], [44], [46], [47]. As we investigate the impact of changes in transcoding parameters, our results can guide better resource utilization for these adaptive video streaming services.

VI. Conclusion

In this paper, we characterize the CPU microarchitectural performance of video transcoding workloads. We vary all the major configurable options of video transcoding operation and explore their impact on microarchitectural performance. We find that most transcoding workloads suffer from back-end issues in the form of high data cache misses. At the same

time, video transcoding operations suffer from instruction cache misses and branch mispredictions in some specific scenarios. To overcome data cache misses, we apply polyhedral optimizer, Graphite on transcoding workloads, and achieve 4.42% average speedup. We show that the state-of-the-art profile-guided optimization technique, AutoFDO can reduce instruction cache misses and branch mispredictions of video transcoding workloads to provide 4.66% average speedup. Finally, keeping the bottleneck diversity in mind, we propose a smart scheduler that assigns the best microarchitectural configuration for different transcoding tasks. On average, our proposed scheduler outperforms the random scheduler by 3.72%, and matches with the best scheduler 75% of cases.

ACKNOWLEDGMENTS

We thank anonymous reviewers for their insightful feedback and suggestions. This work was supported by the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA. We thank the University of Michigan EECS 582 course staff as this work was performed by Yuhan Chen and Jingyuan Zhu as their course project.

REFERENCES

[1] About ffmpeg. [Online]. Available: https://www.ffmpeg.org/about.html

[2] Amazon elastic transcoder pricing. [Online]. Available: https://aws.amazon.com/elastictranscoder/pricing/?nc1=h_ls

[3] Big buck bunny about page. [Online]. Available: https://peach.blender.org/about/

[4] Cisco visual networking index (vni) complete forecast update, 2017–2022. [Online]. Available: https://www.cisco.com/c/dam/m/en_us/network-intelligence/service-provider/digital-transformation/knowledge-network-webinars/pdfs/1213-business-services-ckn.pdf

[5] Cpu or gpu: Which processing power you should boost to improve transcoding speed. [Online]. Available: https://pomfort.com/article/which-processing-power-to-boost-for-which-camera-format-cpu-or-gpu/

[6] Encoding presets for x264. [Online]. Available: https://dev.beandog.org/x264_preset_reference.html

[7] Hours of video uploaded to youtube every minute as of may 2019. [Online]. Available: https://www.statista.com/statistics/259477/hours-of-video-uploaded-to-youtube-every-minute/

[8] Intel® vtune™ profiler. [Online]. Available: https://software.intel.com/content/www/us/en/develop/tools/vtune-profiler.html

[9] Linux encoding. [Online]. Available: https://sites.google.com/site/linuxencoding/x264-ffmpeg-mapping

[10] Perf wiki. [Online]. Available: https://perf.wiki.kernel.org/index.php

[11] Understanding how general exploration works in intel® vtune™ amplifier. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/understanding-how-general-exploration-works-in-intel-vtune-amplifier-xe.html

[12] Understanding rate control modes (x264, x265, vpx). [Online]. Available: https://slhck.info/video/2017/03/01/rate-control.html

[13] Video compression picture types. [Online]. Available: https://en.wikipedia.org/wiki/Video_compression_picture_types

[14] "Video space calculator," https://www.digitalrebellion.com/webapps/videocalc?format=uncompressed_8_1080&frame_rate=f30&length=1&length_type=seconds.

[15] [x264-devel] making sense out of x264 rate control methods. [Online]. Available: https://mailman.videolan.org/pipermail/x264-devel/2010-February/006934.html

[16] x264 homepage. [Online]. Available: http://www.videolan.org/developers/x264.html

[17] A. Albanese, P. S. Crosta, C. Meani, and P. Paglierani, "Gpu-accelerated video transcoding unit for multi-access edge computing scenarios," in *Proceeding of ICN*, 2017.

[18] A. Balachandran, V. Sekar, A. Akella, S. Seshan, I. Stoica, and H. Zhang, "Developing a predictive model of quality of experience for internet video," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 339–350, 2013.

[19] G. Barlas, "Cluster-based optimized parallel video transcoding," in *Parallel Computing*. Elsevier, 2012, pp. 226–244. [Online]. Available: https://doi.org/10.1016/j.parco.2012.02.001

[20] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Transactions on Architecture and Code Optimization (TACO)*, 2014.

[21] D. Chen, T. Moseley, and D. X. Li, "Autofdo: Automatic feedback-directed optimization for warehouse-scale applications," in *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2016, pp. 12–23.

[22] K. Choi, K. Dantu, W.-C. Cheng, and M. Pedram, "Frame-based dynamic voltage and frequency scaling for a mpeg decoder," in *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, 2002, pp. 732–737.

[23] J. De Cock, A. Mavlankar, A. Moorthy, and A. Aaron, "A large-scale video codec comparison of x264, x265 and libvpx for practical vod applications," in *Applications of Digital Image Processing XXXIX*, vol. 9971. International Society for Optics and Photonics, 2016, p. 997116.

[24] R. K. S. Dilip Kumar Krishnappa, Michael Zink, "Optimizing the video transcoding workflow in content delivery networks," in *Proceedings of the 6th ACM Multimedia Systems Conference*. ACM, 2015, pp. 37–48. [Online]. Available: https://doi.org/10.1145/2713168.2713175

[25] A. Ganjam, F. Siddiqui, J. Zhan, X. Liu, I. Stoica, J. Jiang, V. Sekar, and H. Zhang, "C3: Internet-scale control plane for video quality optimization," in *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, 2015, pp. 131–144.

[26] Y. Huang, A. V. Tran, and Y. Wang, "A workload prediction model for decoding mpeg video and its application to workload-scalable transcoding," in *Proceedings of the 15th ACM international conference on Multimedia*, 2007, pp. 952–961.

[27] Y.-W. Huang, B.-Y. Hsieh, T.-C. Chen, and L.-G. Chen, "Analysis, fast algorithm, and vlsi architecture design for h. 264/avc intra frame coder," *IEEE Transactions on Circuits and systems for Video Technology*, vol. 15, no. 3, pp. 378–401, 2005.

[28] J. Jiang, V. Sekar, H. Milner, D. Shepherd, I. Stoica, and H. Zhang, "{CFA}: A practical prediction system for video qoe optimization," in *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, 2016, pp. 137–150.

[29] S. Lederer, C. Müller, and C. Timmerer, "Dynamic adaptive streaming over http dataset," in *Proceedings of the 3rd multimedia systems conference*, 2012, pp. 89–94.

[30] Z. Li, Y. Huang, G. Liu, F. Wang, Z.-L. Zhang, and Y. Dai, "Cloud transcoder: Bridging the format and resolution gap between internet videos and mobile devices," in *Proceedings of the 22nd international workshop on Network and Operating System Support for Digital Audio and Video*, 2012, pp. 33–38.

[31] C.-W. Lin and Y.-R. Lee, "Fast algorithms for dct-domain video transcoding," in *Proceedings 2001 International Conference on Image Processing (Cat. No. 01CH37205)*, vol. 1. IEEE, 2001, pp. 421–424.

[32] S. Lin, X. Zhang, Q. Yu, H. Qi, and S. Ma, "Parallelizing video transcoding with load balancing on cloud computing," in *2013 IEEE International Symposium on Circuits and Systems (ISCAS2013)*. IEEE, 2013, pp. 2864–2867.

[33] Y. Liu, J. Geurts, J.-C. Point, S. Lederer, B. Rainer, C. Müller, C. Timmerer, and H. Hellwagner, "Dynamic adaptive streaming over ccn: A caching and overhead analysis," in *2013 IEEE international conference on communications (ICC)*. IEEE, 2013, pp. 3629–3633.

[34] A. Lottarini, A. Ramirez, J. Coburn, M. A. Kim, P. Ranganathan, D. Stodolsky, and M. Wachsler, "Vbench: Benchmarking video transcoding in the cloud," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 797–809. [Online]. Available: https://doi.org/10.1145/3173162.3173207

[35] A. Mazumdar, B. Haynes, M. Balazinska, L. Ceze, A. Cheung, and M. Oskin, "Perceptual compression for video storage and processing systems," in *Proceedings of the ACM Symposium on Cloud Computing*, 2019, pp. 179–192.

[36] L. Merritt, "X264: A high performance h.264/avc encoder," 2006.

[37] S. Pop, A. Cohen, C. Bastoul, S. Girbal, G.-A. Silber, and N. Vasilache, "Graphite: Polyhedral analyses and optimizations for gcc," in *Proceedings of the 2006 GCC Developers Summit*. Citeseer, 2006, p. 2006.

[38] Y. Sun, X. Yin, J. Jiang, V. Sekar, F. Lin, N. Wang, T. Liu, and B. Sinopoli, "Cs2p: Improving video bitrate selection and adaptation with data-driven throughput prediction," in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016, pp. 272–285.

[39] M. Sung, M. Kim, M. Kim, and W. W. Ro, "Accelerating hevc transcoder by exploiting decoded quadtree," in *The 18th IEEE International Symposium on Consumer Electronics (ISCE 2014)*. IEEE, 2014, pp. 1–2.

[40] L. Wei, J. Cai, C. H. Foh, and B. He, "Qos-aware resource allocation for video transcoding in clouds," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 27, no. 1, pp. 49–61, 2017.

[41] J. Wen, M. Luttrell, and J. Villasenor, "Trellis-based rd optimal quantization in h. 263+," *IEEE Transactions on Image Processing*, vol. 9, no. 8, pp. 1431–1434, 2000.

[42] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, p. 65–76, Apr. 2009. [Online]. Available: https://doi.org/10.1145/1498765.1498785

[43] J. Xin, C.-W. Lin, and M.-T. Sun, "Digital video transcoding," *Proceedings of the IEEE*, vol. 93, no. 1, pp. 84–97, 2005.

[44] F. Y. Yan, H. Ayers, C. Zhu, S. Fouladi, J. Hong, K. Zhang, P. Levis, and K. Winstein, "Learning in situ: a randomized experiment in video streaming," in *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, 2020, pp. 495–511.

[45] A. Yasin, "A top-down method for performance analysis and counters architecture," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2014, pp. 35–44.

[46] X. Yin, A. Jindal, V. Sekar, and B. Sinopoli, "A control-theoretic approach for dynamic adaptive video streaming over http," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015, pp. 325–338.

[47] B. Zhang, X. Jin, S. Ratnasamy, J. Wawrzynek, and E. A. Lee, "Awstream: Adaptive wide-area streaming analytics," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 236–252.

[48] X. Zhang, T. Huang, Y. Tian, M. Geng, S. Ma, and W. Gao, "Fast and efficient transcoding based on low-complexity background modeling and adaptive block classification," *IEEE Transactions on Multimedia*, vol. 15, no. 8, pp. 1769–1785, 2013.

[49] Z. Zhuang and C. Guo, "Building cloud-ready video transcoding system for content delivery networks (cdns)," in *2012 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2012, pp. 2048–2053.