Theory/Practice Transfer Paper

Matriculation number:	
Accepted topic:	
Bachelor‘s programme, centuria:	

Contents

List of Figures

1 Introduction

In the development of software applications, it is common to do automated UI testing. For web-applications, this is usually done by automating the browser with tools like Puppeteer [1] or WebDrivers [2]. With projects growing larger the test durations are increasing, which in turn requires a testing solution that can scale horizontally. Parallel testing, however, poses a problem in the traceability of errors since one tester is no longer capable of observing all test executions simultaneously. This raises the requirement for a review at a later time which is usually solved by doing a screen recording. Video recordings, however, are typically very resource-intensive due to the amount of data that needs to be processed and compressed. This increases the number of hardware resources required significantly, effectively reducing the degree of parallelization or increasing the budget requirements. This leaves the question of which codecs are the fastest and most efficient in terms of CPU usage while still retaining an acceptable compression ratio to preserve network and disk bandwidth.

Given that question, this work will focus on two aspects of the delivery of screen recordings from parallelized, automated Web-UI tests:

1. Which video codecs have a positive impact on resource usage during screen capture?
2. How can screen recordings be efficiently delivered to the tester?

Like many companies, PPI AG has multiple projects with web-based frontends that require testing. For most projects, this process has already been automated but either parallelization or visual feedback is still lacking. This analysis can aid the process of cost-effectively parallelizing the current testing infrastructure while retaining the visual feedback test operators are used to.

To answer the initial question this document will be split into three parts. Firstly, the empirical method that will be used to compare different codecs will be outlined. The second part uses the described methods to measure and analyze the data. Finally, different technologies and streaming formats that can be used to deliver the screen recording will be compared by taking a look at their format specifications.

2 Methodology

In order to ensure the reproducibility and to extend the applicability of the result to the broadest possible set of scenarios, a clear methodology has to be defined. This includes criteria like the screen content, effects on compression, browser use, web pages, how measurements are taken and later evaluated. Many of these can be choosen arbitrarily and decisions have been made based on the environment this research originates from. Others are backed by public research and usage numbers.

2.1 Screen content

To evaluate the resource consumption of different codecs and parameters, a non-empty[1], reproducible, and realistic screen content is required. Since most tests at PPI AG require a complex environment, this research will make use of a tailored screen content that closely resembles a real-world testing environment.

Compression

Most codecs make use of compression algorithms to save bandwidth. These algorithms behave differently on static or very similar content than they do on very detailed, moving contents with the impact on CPU and bitrate depending on the codec used [3]. To rule out this interference which would not be present in a real-world test scenario this research is going to use an automated browser instance that covers the whole screen. This however will have some implications on the applicability of the results — more details can be found in section 5.1.

Browser

All browsers are expected to render web-pages the same way and the results should be reproducible independently of the browser chosen. Thus, the project will make use of the desktop browser with the highest market share as of March 2020, which according to the data in figure 5 is Google Chrome. It will be automated by using the Python Selenium library[2] which interacts with the browser through a REST API provided by the vendor[3]. The screen resolution will be set to `1920x1080` which is the most frequently used desktop resolution in Germany over the past 12 months as seen in figure 6.

Web pages

During the recording, the browser will be loading multiple web-pages and jump to different scroll positions to resemble a realistic test-workload. The pages contain a combination of static and dynamic content together with some animations. To automate this process the previously mentioned Selenium library will be used. The implementation can be found in section B.2.

[1] An empty screen would skew the results since compression algorithms do not have to work as hard
[2] Available at https://github.com/SeleniumHQ/selenium
[3] More details available at https://selenium.dev/

2.2 Measurement

To accurately capture the CPU load created by the encoding process the CPU time will be used as a measurement value. It represents the amount of time a process is actively running instructions on the CPU. In comparison to the regular duration the process ran, it takes scheduling into account and prevents any interruptions of the process from affecting the final result. This value can be further divided into the time spent in the so-called user-space and system-space. The former includes all instructions and calculations the program did (e.g. compression) while the latter includes all instructions performed by the operating system on behalf of the process like for example disk read/write operations. The regular duration and file size of the resulting video will be recorded as well. The codecs will be measured one after another with a pause in between each evaluation. This pause is employed to prevent any influence that other mechanisms may have like swapping or thermal load.

Recording framework

A cross-platform program to record videos called FFMpeg will be used as an encoding tool with the X11grab plugin which records the screen contents on Linux by requesting the framebuffer contents from the window server. All resulting frames will be written to the codecs recommended container format. Alternatively, a low overhead container called NUT will be used if no recommended format is available. Additionally, the process will be restricted to a single processor core using Linux control groups to recreate an environment with restricted resources.

Specifications of the test device

All measurements will be recorded on a 2019 16-inch MacBook Pro with 16 GB of RAM, 1 TB SSD, and the Intel Core i9-9880H CPU running macOS 10.15.4. All non-essential applications will be closed during the recordings to prevent any interference. Additionally, the device will be attached to a power outlet and sufficient cooling will be provided to prevent excessive CPU throttling.

Context and sampling

To recreate an environment that closely resembles a real-world scenario the recordings will be performed within Docker containers described in detail in section 2.2. This, however, may yield inconsistencies on macOS since Docker makes use of a virtual machine [4]. To smooth out any influence this may have each codec will be evaluated multiple times and the median will be used as the final metric. To determine a reasonable sample size the test methodology will be executed multiple times with varying sample sizes and the resulting confidence interval calculated. From there a reasonable sample size will be chosen.

Docker environment

By default, most Docker containers are headless and do not contain a desktop environment required for both screen recording and browser automation. A full desktop environment comes with a lot of overhead in terms of window management, rendering, and additional services required. To reduce this overhead as much as possible, a virtual implementation of the X11 protocol will be used that provides a virtual, in-memory framebuffer which is not backed by any graphics devices (more specifically Xvfb). This choice could have positive effects on the CPU usage claimed by the importer, which reads frames

from the framebuffer, and since the importer and exporter can not be measured independently it should give more accurate results.

3 Evaluation

The following section contains the execution of the methodology described in section 2. Codecs have been chosen based on their maturity and software support however some emerging ones have been included as well.

3.1 Codecs and Encoders

The first codec to include consist of a raw data dump directly from the framebuffer to establish a baseline to compare against. This codec is called `rawvideo` in the used recording framework. The second codec, H.264, has been selected due to its wide support in both software and hardware in addition to the maturity and degree of optimization that available encoders have. Two versions of the encoder are available where one uses the RGB pixel format (the framebuffers native format) and the other the YC_BC_R format which is more common in video processing pipelines - both versions will be evaluated. For this evaluation the `libx264` implementation of the codec by VideoLAN will be used as multiple independent tests have shown it outperforming other implementations by as much as 24% [5] and it is freely available.

The successor of H.264, HEVC/H.265, will also be included since it promises better video quality [6] at the same compression levels and similar to its predecessor both software and hardware implementations are around the corner for many platforms. Again, the implementation by VideoLAN called `libx265` will be used for the same reasons as with H.264.

Another common codec [7] and a competitor to the two aforementioned ones is VP9 which is developed by Google Inc. [8] and used by YouTube [9] and other streaming platforms like Netflix [10] which makes it a mentionable contender. The reference implementation by Google, published under the WebM project, called `libvpx` will be used.

One codec which is heavily in use for proxy media[4] and final video delivery[5] called ProRes will be included for its optimizations regarding the use as intermediate media [11] as well as another broadcasting related codec developed by the BBC called VC2[6] which has the potential for efficient encoding of screen content due to its low overhead and low latency design [12].

A recent innovation in the media industry is the AV1 codec which promises even higher compression ratios than VP9 or H.265 at a comparable visual quality [13]. However, running a test using the publicly available and self-proclaimed “fastest” encoder available called rav1e yielded very poor performance even across 16 threads taking more than three seconds per frame at full CPU utilization[7]. That makes the codec unviable for use in a test-recording environment where resources are at a premium and thus it will be excluded from the test.

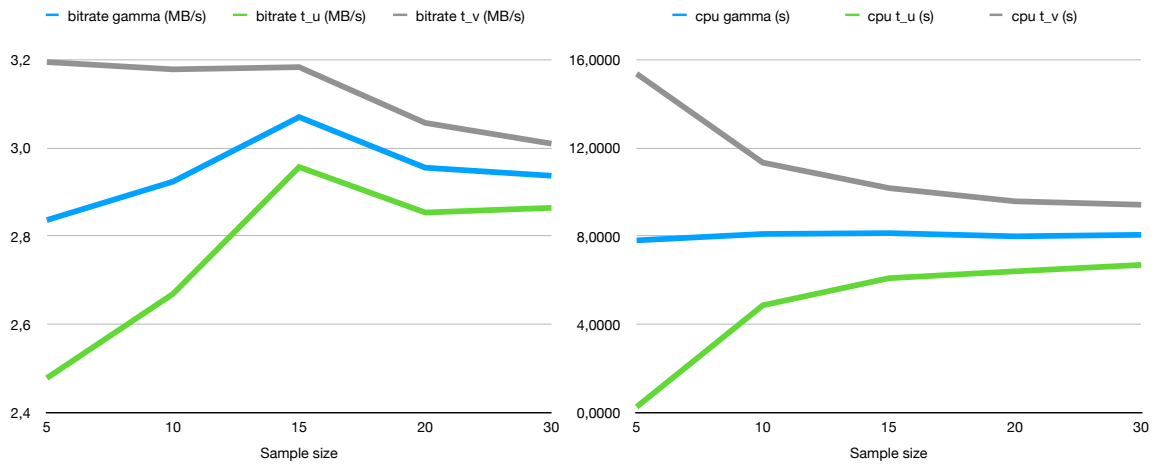
[4] intermediate video files used for faster editing in visual effects workflows
[5] e.g. on BluRays
[6] formerly called Dirac
[7] Note that this situation may change very fast due to the ongoing development of encoders [14]

In summary, the following codecs, encoders and container formats will be used:

Codec	Container	Encoder
Raw	nut	-
VC2	nut	reference
VP9	webm	libvpx-vp9
H.264	mp4	x264
H.264	mp4	x264rgb
H.265	mp4	x265
ProRes	mov	prores_ks

3.2 Determining a sample size

Using the method described in section 2.2 measurements for different sample sizes have been made. The resulting confidence intervals can be seen in figure 1 with t_u being the lower bound, t_v the upper bound, and gamma representing the median of the measured values. They exhibit a large variation at sample sizes below 15 and only marginal improvement in interval size beyond 20. To achieve a reasonable trade-off between computing time and statistical relevance a sample size of 20 will be used for all further measurements.



(a) Bitrate (b) CPU time

Figure 1: Confidence intervals

3.3 Measurements

All measurements have been made over a period of just over 3,5 hours and the resulting data processed using a python script which can be found in the accompanying GitHub repository[8] or in the appendix. The resulting medians for bitrates, CPU times and disk I/O have been plotted. The graphs will be analyzed in the following subsections. X-Axis labels contain the codec, compression (L = Lossless, C = Compressed), the container format, and finally the pixel format.

[8] to retain author anonymity the link has been excluded from this version of the document

3.3.1 Bitrate

The data in figure 2 shows the expected difference between compressed and lossless with ProRes and VC2 as outliers that are even higher than lossless versions of most other codecs. This could be attributed to their optimization for the video industry where file size is secondary and decode time matters far more [11].

H.265 exhibits a very low bitrate in comparison to H.264 and VP9 in both compressed and lossless mode. However, the resulting video files have a very low framerate that undercuts the target of 15 FPS by a large margin. This phenomenon has been analyzed further and only exhibits itself when the encoder is restrained to a single core which might indicate a CPU bottleneck or very poor single-core optimization.

The difference between the two H.264 encoders is only marginal in compressed mode while the native pixel format yields about double the rate in lossless mode. The raw dump, on the other hand, shows a bitrate that is significantly higher than all the other codecs at 21 times the bitrate of VC2 making it unfeasible for any kind of short and long term storage[9].

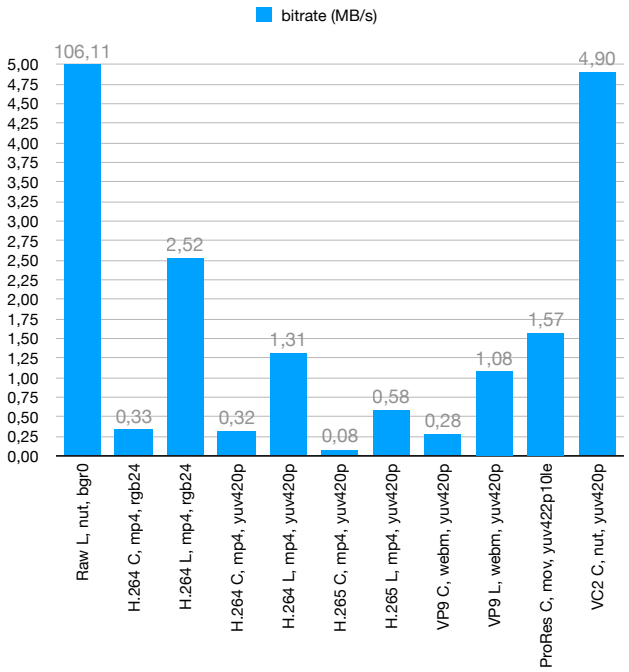


Figure 2: Measurement - Bitrate

[9]Using a technology proposed in 2010 that enables 4.2 Gigabit/s write speeds on SSDs [15] only five parallel recordings would be possible before completely saturating the write bandwidth

3.3.2 CPU

The graph for the CPU in figure 3 shows stacked bars where the blue portion represents the time spent by the process calculating in userspace[10] while the green portion exhibits the time spent in kernel calls[11]. It is notable that while the raw write takes almost no time in user-space, it spends a significant amount of time in system-space which can be attributed to the high amount of write operations performed in comparison to other codecs with a significantly lower bitrate making it almost as slow as lossless H.264 (RGB).

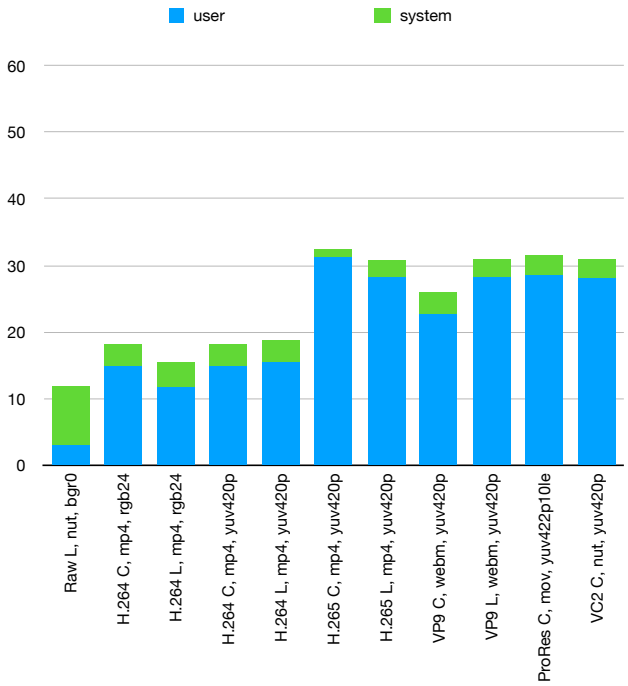


Figure 3: Measurement - CPU time

Also notable is the difference between lossless and compressed codecs with H.264 (RGB) and H.265 behaving as expected with lossless taking less time while others like H.264 (YC_BC_R) and VP9 are contrary to that with compressed being faster. This behavior could simply be attributed to measurement inaccuracy or minor differences in screen content due to page load times.

Another feature of the data is the minimal difference between the two versions of the H.264 encoder with the two compressed configurations using the same amount of CPU (within the margin of error). Differences emerge when using the lossless option where the source pixel format is using slightly fewer resources.

It should also be noted that while the aforementioned issues with H.265 are represented in the bitrate graph by a very low bitrate they are not visible from the CPU usage. The current testing environment restrains all codecs to a single CPU core, however, by removing these constraints and instead of asking the codecs to use just a single core a similar result can be achieved except for H.265 which uses almost 70 seconds of CPU time. This confirms the previously mentioned bottleneck and explains the very low bitrate and framerate under regular test conditions.

[10] e.g. compression of data

[11] e.g. `write(2)` or `read(2)`

4 Distribution

After recording the screen with one of the previously evaluated codecs the files are likely stored in a Storage Area Network [16] and have to be delivered to the person evaluating the test results for review. Multiple methods of delivery exist for protocols like HTTP, RTMP, or even analog broadcasting but since the environment is are already utilizing a web-based platform only protocols in that realm will be considered.

In the domain of plain HTTP, there are many ways of delivering video content and the most used ones according to a reasonably recent survey by bitmovin with 456 participants [7] will be further analyzed.

4.1 HTTP Live Streaming

The most used streaming format is called HTTP Live Streaming or HLS for short. An initial draft was published by Apple in May 2009 and was changed several times throughout the following years until the RFC was officially approved by the Internet Engineering Task Force (IETF) in August 2017 and has been an official standard since [17]. The protocol is natively supported by all devices running iOS 3.0 / Safari 4.0 or later [18].

The format focuses on simple delivery using only a regular HTTP file server[12]. For this, a manifest file has to be generated which contains references to fragments[13] indexed by time offsets to allow fast random access. Supported encodings, however, are restricted to H.264 and H.265 by the standard even though other formats are likely to work if they support the MP4 or MPEG-TS container format.

One advantage of HLS is that it offers both Video on Demand (VOD) and Live delivery modes which allow for both a live view and later availability based on the same file and using the same streaming format with only minor changes to the manifest file.

4.2 MPEG Dynamic adaptive streaming over HTTP

The second format is MPEG Dynamic adaptive streaming over HTTP or MPEG-DASH. The initial draft has been submitted to the ISO in January 2011 and was published in April 2012, following standardization in November 2011.

The format operates very similarly to HLS by segmenting the underlying media into segments of a fixed duration which are referenced by time offsets [19]. Delivery is also very similar to the previous format as only a regular HTTP server is required once the media has been formatted correctly and manifests have been generated. Unlike HLS the specification is fully coding format-agnostic allowing any codec to be used as long as the receiving device can decode it.

Delivery modes are also very similar to HLS with live and on-demand modes available.

[12] e.g. nginx

[13] either individual video files or byte offsets in a single fMP4 file

4.3 MPEG Common Media Application Format

As previously mentioned both HLS and MPEG-DASH are using segmented base media formats indexed by time offsets. This similarity leads to further simplification by using a common media file for both streaming formats with the only difference being the format-specific segment index metadata. This usage of the segmented base media has been standardized by the ISO as the MPEG Common Media Application Format (CMAF) [20]. It further simplifies video delivery to the broadest possible audience since no transmuxing or even transcoding is required when using streaming formats that support CMAF and a wider range of clients can be served with less resource usage.

The figure 4 shows the rough timeline of the CMAF development. Back in 2010 each streaming technology operated independently and required different versions of the same media files in addition to the metadata. In 2016 the underlying media format and segmentation have been standardized and other streaming formats could be derived from this. However, it was still necessary to define all metadata attributes which are not stored in the media tracks seperately. About a year later a common metadata format has been introduced as an extension to the standard which included those attributes. From this point on the manifests for each format can be derived from the shared manifest and only format specific properties need to be set manually to enable multi-format streaming. In theory most formats are still supporting other base media formats (e.g. MPEG-TS for HLS or M2TS for DASH) according to the specifications, however in practice this is rarely used.

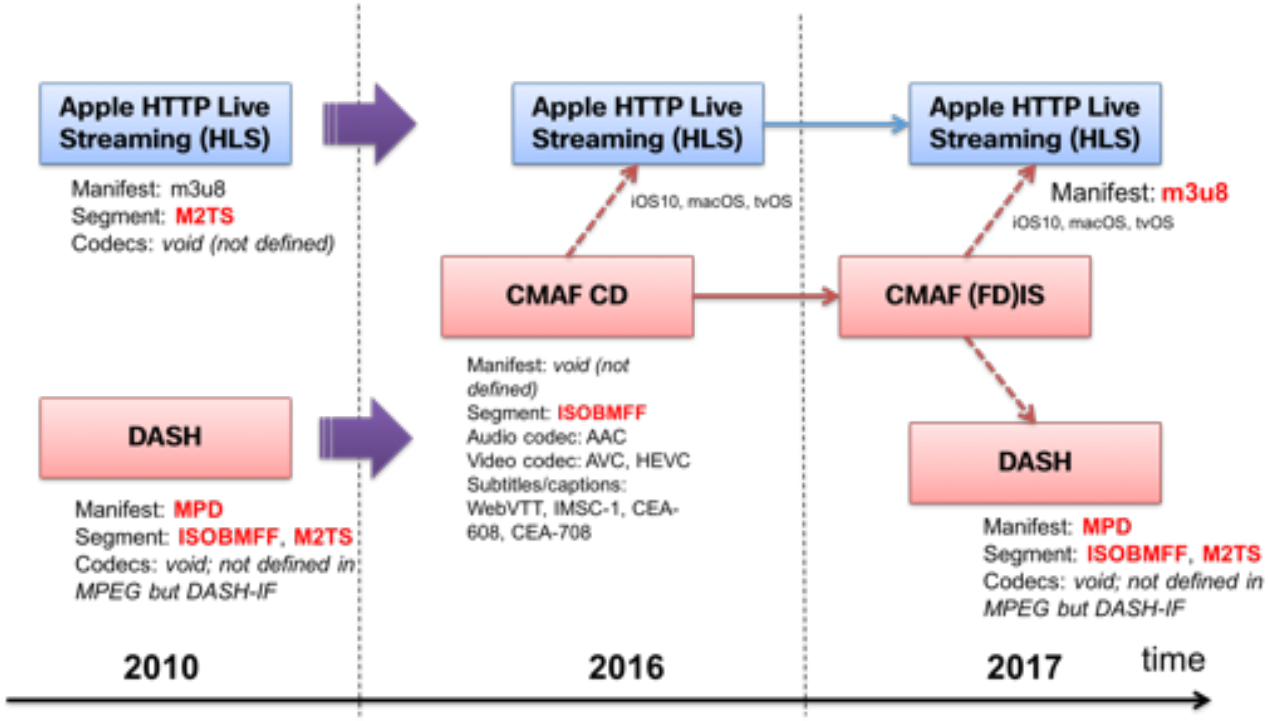


Figure 4: CMAF timeline [21]

5 Conclusion

Overall H.264 turns out to be the most time-efficient codec while still retaining a reasonable bitrate. Even though the codec is close to 17 years old it still provides one of the most mature, configurable

and well-optimized encoders and has very good encoding and decoding support by both software and hardware alike. More modern codecs may have the potential to provide an even lower bitrate at comparable quality as the results showed for both VP9 and H.264. However, that comes with a significant cost in computing resources due to a lack of optimization[14] making potential bitrate savings of up to 50% [6] with more recent codecs unviable.

For screen recordings H.264's space efficiency remains sufficient at just over one Gigabyte per hour especially since most screen recordings will only be viewed over the local network. The pixel format conversion between the source and YC_BC_R takes almost no additional resources when the video is being compressed and since the latter format is required by e.g. HLS doing the conversion in the initial encoding phase makes a secondary conversion step obsolete, effectively saving resources, cost, and reducing complexity.

Thanks to the standardization of a common media fragment format (namely CMAF) it is no longer necessary to decide between different streaming formats and one source file can be used without any format-specific transmuxing or transcoding. Additionally, both live and on-demand delivery is supported by all mentioned streaming formats which makes the delivery of screen recordings simple. It is even possible to generate HLS and MPEG-DASH manifests directly during the capture with the used encoding framework FFMPEG [22] which further reduces the overall system complexity and simplifies the storage and delivery of screen recordings.

5.1 Applicability

Various factors may limit the applicability of the results obtained. They are tied to the specific environment they have been recorded in and different CPUs, operating systems and other variations may shift the absolute numbers due to higher or lower computing capability or hardware optimizations. For this reason only the relative difference between the numbers should be used directly and absolute numbers should be re-evaluated within the target environment.

In addition, various codecs may react differently to varying screen contents. This may change the results significantly depending on the screen contents as various compression algorithms operate differently and thus may use varying amounts of computing resources and yield differing compression ratios [3]. For this reason only applications where similar screen contents are used should incorporate the results directly into decisions. For other scenarios a re-evaluation may be necessary.

5.2 Further research

This work specifically excluded hardware-accelerated encoding due to the limited availability and special hardware requirements. However, especially for very common codecs like H.264 and H.265, it has the potential to provide significant speed benefits while delivering higher quality [23] making it interesting for further research.

Additionally, a more complex two-stage compression method could be evaluated where the first stage makes use of a very fast codec that provides reasonable space efficiency but does not need to fit into the playback constraints set by e.g. HLS. A second stage would then compress the video even further

[14] this may change in the future

and convert it to the target format on a different CPU to prevent interference with the test. This method could provide even greater bandwidth savings when storage space is limited.

Literature

[1] “Pupeteer.” [Online]. Available: https://pptr.dev/#?product=Puppeteer&version=v3.0.1. [Accessed: 20-Mar-2020].

[2] “WebDriver.” [Online]. Available: https://w3c.github.io/webdriver. [Accessed: 10-Feb-2020].

[3] J.-R. Ohm, G. J. Sullivan, H. Schwarz, T. K. Tan, and T. Wiegand, “Comparison of the coding efficiency of video coding standards—including high efficiency video coding (hevc),” *IEEE Transactions on circuits and systems for video technology*, vol. 22, no. 12, pp. 1669–1684, 2012.

[4] “Announcing LinuxKit: A Toolkit for building Secure, Lean and Portable Linux Subsystems,” Apr-2018. [Online]. Available: https://www.docker.com/blog/introducing-linuxkit-container-os-toolkit/. [Accessed: 18-Mar-2020].

[5] D. Vatolin, D. Kulikov, and A. Parshin, “Sixth MPEG-4 AVC/H.264 Video Codecs Comparison,” 25-May-2010. [Online]. Available: http://www.compression.ru/video/codec_comparison/h264_2010/. [Accessed: 01-Jun-2020].

[6] J. D. 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*, 2016, vol. 9971, pp. 363–379.

[7] Bitmovin, “Video developer report,” 2018. [Online]. Available: https://go.bitmovin.com/hubfs/Bitmovin-Video-Developer-Report-2018.pdf. [Accessed: 17-Mar-2020].

[8] A. Grange, P. de Rivaz, and J. Hunt, “VP9 bitstream & decoding process specification version 0.6,” 2016. [Online]. Available: https://storage.googleapis.com/downloads.webmproject.org/docs/vp9/vp9-bitstream-specification-v0.6-20160331-draft.pdf. [Accessed: 01-Apr-2020].

[9] S. Robertson, “VP9: Faster, better, buffer-free youtube videos,” Apr-2015. [Online]. Available: https://youtube-eng.googleblog.com/2015/04/vp9-faster-better-buffer-free-youtube.html. [Accessed: 28-Mar-2020].

[10] M. Manohara, A. Moorthy, J. D. Cock, I. Katsavounidis, and A. Aaron, “Optimized shot-based encodes: Now streaming!” Mar-2018. [Online]. Available: https://netflixtechblog.com/optimized-shot-based-encodes-now-streaming-4b9464204830. [Accessed: 02-Apr-2020].

[11] Apple, “Apple ProRes Whitepaper,” Jan-2020. [Online]. Available: https://www.apple.com/final-cut-pro/docs/Apple_ProRes.pdf. [Accessed: 01-Apr-2020].

[12] BBC, “Dirac specification dirac specification version 2.2.3,” Sep-2008. [Online]. Available: https://web.archive.org/web/20141006092456/http://diracvideo.org/download/specification/dirac-spec-latest.pdf. [Accessed: 25-Mar-2020].

[13] Y. Liu, “AV1 beats x264 and libvpx-vp9 in practical use case,” Apr-2018. [Online]. Available: https://engineering.fb.com/video-engineering/av1-beats-x264-and-libvpx-vp9-in-practical-use-case/. [Accessed: 29-Mar-2020].

[14] N. Egge and B. Vibber, “AV1: One year later,” Mile High Video, Jul. 2019.

[15] T. Hatanaka *et al.*, “A 60% higher write speed, 4.2gbps, 24-channel 3D-solid state drive (ssd) with nand flash channel number detector and intelligent program-voltage booster,” in *2010 symposium on vlsi circuits*, 2010, pp. 233–234.

[16] R. S. Arunkundram and P. Sachdev, *Using storage area networks (special edition)*. Que, 2001.

[17] R. Pantos and W. May, "HTTP Live Streaming," Aug-2017. [Online]. Available: https://rfc-editor.org/rfc/rfc8216.txt. [Accessed: 04-Apr-2020].

[18] Apple, "Understanding the http live streaming architecture." [Online]. Available: https://developer.apple.com/documentation/http_live_streaming/understanding_the_http_live_streaming_architecture. [Accessed: 20-Mar-2020].

[19] ISO/IEC JTC 1/SC 29 Coding of audio, picture, multimedia and hypermedia information, "Information technology — Dynamic adaptive streaming over HTTP (DASH) — Part 1: Media presentation description and segment formats," International Organization for Standardization, Geneva, CH, Standard, Dec. 2020.

[20] ISO/IEC JTC 1/SC 29 Coding of audio, picture, multimedia and hypermedia information, "Information technology — Multimedia application format (MPEG-A) — Part 19: Common media application format (CMAF) for segmented media," International Organization for Standardization, Geneva, CH, Standard, Mar. 2020.

[21] C. Timmerer, "MPEG-CMAF: Threat or Opportunity?" 04-Apr-2016. [Online]. Available: https://bitmovin.com/what-is-cmaf-threat-opportunity/. [Accessed: 06-Jun-2020].

[22] S. Sabatini, "FFmpeg formats documentation." [Online]. Available: https://ffmpeg.org/ffmpeg-formats.html#segment_002c-stream_005fsegment_002c-ssegment. [Accessed: 06-Apr-2020].

[23] M. A. Wilhelmsen, H. K. Stensland, V. R. Gaddam, P. Halvorsen, and C. Griwodz, "Performance and Application of the NVIDIA NVENC H. 264 Encoder."

[24] "Browser market share worldwide," Mar-2019. [Online]. Available: https://gs.statcounter.com/browser-market-share. [Accessed: 15-Mar-2020].

[25] "Desktop screen resolution stats germany," Mar-2019. [Online]. Available: https://gs.statcounter.com/screen-resolution-stats/desktop/germany. [Accessed: 15-Mar-2020].

Appendices

A Figures

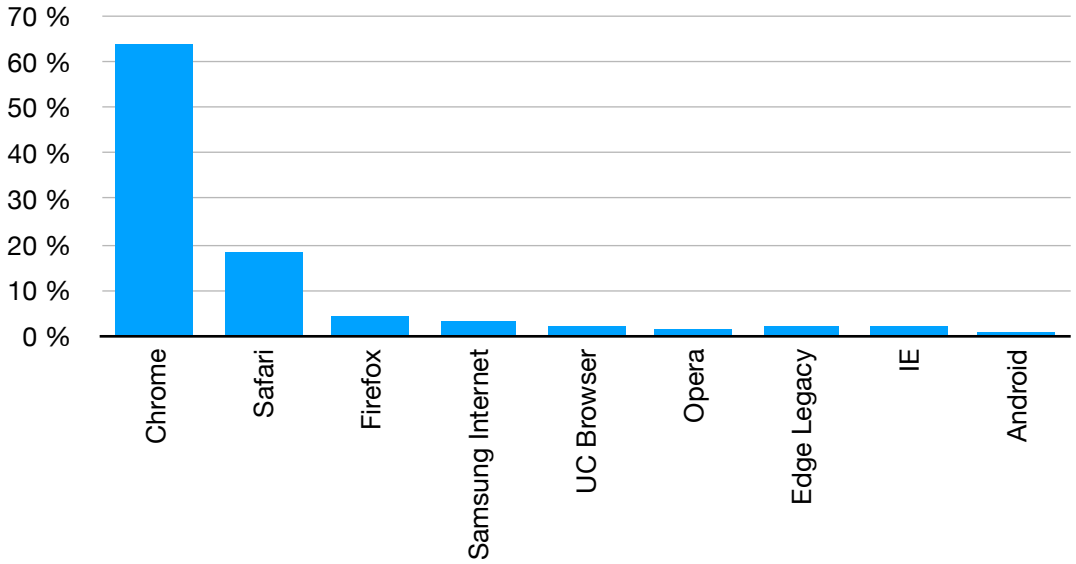


Figure 5: Global browser marketshare (March 2019) [24]

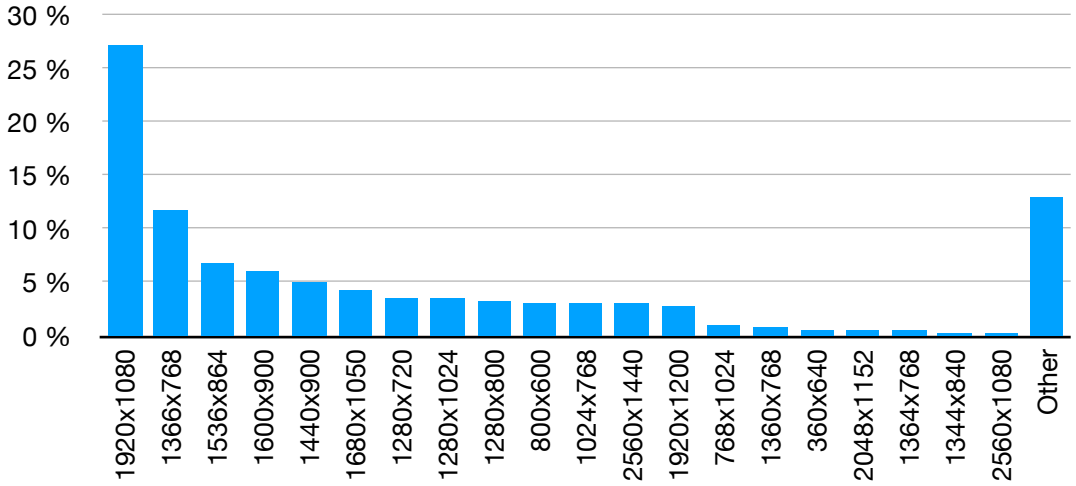


Figure 6: Screen resolutions in Germany (March 2019) [25]

B Code listings

Below is a list of parameters and python scripts that have been used to take all measurements. Additionally the Dockerfile containing the desktop environment and browser has been included. All files listed below can be accessed online at GitHub, however the link has been excluded from this version of the document to retain author anonymity.

B.1 Codec parameters

The following list has been used to execute the test runs and each cmdline was combined with the common prefix `ffmpeg -y -rtbufsize 1500M -probesize 100M -framerate 15 -video_size 1920x1080 -f x11grab -i :42 -t 00:01:00 -threads 1`.


```
{
    "name": "Raw",
    "pixel_format": "bgr0",
    "compression": false,
    "container": "nut",
    "cmdline": "-c:v rawvideo -pix_fmt bgr0"
},

{
    "name": "H.264",
    "pixel_format": "rgb24",
    "compression": false,
    "container": "mp4",
    "cmdline": "-c:v libx264rgb -preset ultrafast -crf 0 -tune stillimage -x264-params keyint=30:scenecut=0:keyint_min=30 -f mp4"
},
{
    "name": "H.264",
    "pixel_format": "rgb24",
    "compression": true,
    "container": "mp4",
    "cmdline": "-c:v libx264 -preset ultrafast -crf 28 -tune stillimage -x264-params keyint=30:scenecut=0:keyint_min=30 -f mp4"
},

{
    "name": "H.264",
    "pixel_format": "yuv420p",
    "compression": false,
    "container": "mp4",
    "cmdline": "-c:v libx264 -preset ultrafast -crf 0 -tune stillimage -x264-params keyint=30:scenecut=0:keyint_min=30 -f mp4"
},
{
    "name": "H.264",
    "pixel_format": "yuv420p",
    "compression": true,
    "container": "mp4",
    "cmdline": "-c:v libx264 -preset ultrafast -crf 28 -tune stillimage -x264-params keyint=30:scenecut=0:keyint_min=30 -f mp4"
},

{
    "name": "H.265",
    "pixel_format": "yuv420p",
    "compression": false,
    "container": "mp4",
    "cmdline": "-c:v libx265 -preset ultrafast -x265-params lossless=1 -pix_fmt yuv420p -f mp4"
},
{
    "name": "H.265",
    "pixel_format": "yuv420p",
    "compression": true,
    "container": "mp4",
    "cmdline": "-c:v libx265 -preset ultrafast -crf 28 -pix_fmt yuv420p -f mp4"
},

{
    "name": "VP9",
    "pixel_format": "yuv420p",
    "compression": false,
    "container": "webm",
    "cmdline": "-c:v libvpx-vp9 -lossless 1 -cpu-used 8 -deadline realtime -pix_fmt yuv420p"
```

```
    },
    {
        "name": "VP9",
        "pixel_format": "yuv420p",
        "compression": true,
        "container": "webm",
        "cmdline": "-c:v libvpx-vp9 -crf 35 -b:v 0 -cpu-used 8 -deadline realtime -pix_fmt yuv420p"
    },

    {
        "name": "ProRes",
        "pixel_format": "yuv422p10le",
        "compression": true,
        "container": "mov",
        "cmdline": "-c:v prores_ks -profile:v 0 -qscale:v 13 -pix_fmt yuv422p10le"
    },

    {
        "name": "VC2",
        "pixel_format": "yuv420p",
        "compression": true,
        "container": "nut",
        "cmdline": "-c:v vc2 -wavelet_depth 1 -b 320K -pix_fmt yuv420p"
    }
```

B.2 Browser control code and CPU measuring code

The following file is executed within the Docker container to record the screen, take measurements and automate the browser. It is called `evaluate.py` on disk.

```
#!/usr/bin/env python3
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
from time import sleep, time
from scipy.interpolate import interp1d
from threading import Thread
import numpy
import os
import csv
import json
import subprocess
import psutil

display = ":42"
cmdline_codec = os.environ['CODEC_CMDLINE']
cmdline = "ffmpeg -y -rtbufsize 1500M -probesize 100M -framerate 15 -video_size 1920x1080 -f x11grab -i "
    + display +
    " -t 00:01:00 -threads 1 "
    + cmdline_codec
driver = webdriver.Firefox()

sites = [
    # Regular websites
    "http://www.python.org",

    # Stress the codec with particles
    "https://vincentgarreau.com/particles.js/#nasa",
    "https://codepen.io/themegatb/full/VwvaNpv",
    "https://vincentgarreau.com/particles.js/#nyancat2",

    # Some UI animations
    "https://www.highcharts.com/demo",
```

```
    "https://www.highcharts.com/demo/column-stacked",
    "https://www.highcharts.com/demo/column-parsed"
]

def scroll_script(speed):
    return """
        function scroll() {{
            document.scrollingElement.scrollTop += {step};
            window.requestAnimationFrame(scroll);
        }}

        scroll()
    """.format(step=speed)

def scroll(speed):
    sleep(2)
    driver.execute_script(scroll_script(speed))

def setup_browser():
    driver.set_window_size(1920, 1080)

def do_browser_stuff():
    # Iterate over all the pages
    for site in sites:
        driver.get(site)
        sleep(3)

    # Finish off with some scrolling on a heavily scroll-animated and "contentful" web-page
    driver.get("https://www.apple.com/de/macbook-pro-16/")
    scroll(20)

def read_bitrate(path):
    ffprobe_cmd = ("ffprobe -v error -select_streams v:0 -show_entries format=bit_rate -of default=noprint_wrappers=1:nokey=1 " + path).split(" ")
    ffprobe = subprocess.Popen(ffprobe_cmd, stdout=subprocess.PIPE, stderr=subprocess.STDOUT)
    ffprobe.wait()
    stdout, stderr = ffprobe.communicate()

    # If no bitrate is available (e.g. raw video)
    if stdout == b'N/A\n':
        stdout = b'-1'

    return int(stdout)

def record():
    animator = Thread(target=do_browser_stuff)
    ffmpeg = subprocess.Popen(cmdline.split(" "), stdout=subprocess.PIPE, stderr=subprocess.STDOUT)
    process = psutil.Process(ffmpeg.pid)

    animator.start()

    recordingInfo = {
        "bitrate": 0,
        "memory": [],
        "times": {
            "user": [],
            "system": [],
            "wall": []
        },
        "io": {
            "read": [],
            "write": []
        }
    }

    while ffmpeg.poll() is None:
        # Don't use too much CPU
```

```
        sleep(0.25)

        # Grab some data from the process
        with process.oneshot():
            memory = process.memory_info().rss / 1024 / 1024
            times = process.cpu_times()
            timestamp = round(time() - process.create_time(), 4)
            io = process.io_counters()

            recordingInfo['memory'].append(memory)

            recordingInfo['times']['user'].append(times.user)
            recordingInfo['times']['system'].append(times.system)
            recordingInfo['times']['wall'].append(timestamp)

            recordingInfo['io']['read'].append(io.read_chars)
            recordingInfo['io']['write'].append(io.write_chars)

    process.wait()
    animator.join()

    recordingInfo['bitrate'] = read_bitrate(cmdline.split(" ")[-1])

    stdout, stderr = ffmpeg.communicate()
    with open('/out/log.txt', 'ab') as log:
        log.write(stdout)

    return recordingInfo

print("\t" + cmdline, flush=True)
setup_browser()
measurements = []

sample_size = int(os.environ.get('SAMPLE_SIZE', "20"))

for i in range(0, sample_size):
    print("\tBuild " + str(i + 1) + "/" + str(sample_size), flush=True)
    measurements.append(record())

result = {
    "cmdline": cmdline,
    "measurements": measurements,
}

with open('/out/data.json', 'w') as file:
    json.dump(result, file, sort_keys=True)
```

B.3 Dockerfile

This Dockerfile creates a virtual desktop environment, downloads the remote control driver and its dependencies and installs various python packages. At runtime it starts the virtual framebuffer and launches the evaluation script found under section B.2.

```
FROM alpine

RUN apk add --update --no-cache xvfb ffmpeg firefox-esr ttf-dejavu && rm -rf /var/cache/apk/*

RUN wget -q -O /etc/apk/keys/sgerrand.rsa.pub https://alpine-pkgs.sgerrand.com/sgerrand.rsa.pub && \
    wget https://github.com/sgerrand/alpine-pkg-glibc/releases/download/2.30-r0/glibc-2.30-r0.apk && \
    wget https://github.com/sgerrand/alpine-pkg-glibc/releases/download/2.30-r0/glibc-bin-2.30-r0.apk && \
    apk add glibc-2.30-r0.apk glibc-bin-2.30-r0.apk
```

```
RUN wget https://github.com/mozilla/geckodriver/releases/download/v0.26.0/geckodriver-v0.26.0-linux64.tar.gz && \
    tar xzvf geckodriver*.tar.gz && \
    chmod +x geckodriver && \
    mv geckodriver /usr/local/bin

RUN apk add --update --no-cache linux-headers musl-dev gcc python3-dev python3 py3-pip py3-scipy py3-psutil py3-numpy
RUN pip3 install selenium requests

ENV DISPLAY=:42

RUN echo '(Xvfb $DISPLAY -ac -wr +render -noreset +extension GLX -screen 0 1920x1080x24 >/dev/null 2>&1 &) && sleep 2 && /evaluate.py' >/entrypoint.sh
COPY evaluate.py /
CMD /entrypoint.sh
```

B.4 Measurement execution

The following two scripts have been employed to launch docker containers and capture the resulting data for regular measurements and the initial variation check. The first script accesses the codec list defined in section B.1 as `codecs.json`.

measure.py

```
#!/usr/bin/env python3
import json
import os
import shutil
import subprocess

with open('codecs.json') as json_data:
    codecs = json.load(json_data)

tmp_path = "/tmp/encodings"
output_path = "out"

if not os.path.exists(output_path):
    os.makedirs(output_path)

for i, codec in enumerate(codecs):
    cmdline = codec['cmdline'] + " /out/video." + codec['container']
    compression = "C" if codec['compression'] else "L"
    parameters = compression + ", " + codec['container'] + ", " + codec['pixel_format']

    print(codec['name'] + "\t(" + parameters + ") [" + str(i + 1) + "/" + str(len(codecs)) + "]")

    if not os.path.exists(tmp_path):
        os.makedirs(tmp_path)

    docker_cmd = [
        "docker", "run",
        "--rm", "-it",
        "--name", "encodings",
        "--shm-size", "2g",
        "--cpuset-cpus", "0",
        "-v", tmp_path + ":/out",
        "-e", "CODEC_CMDLINE=" + cmdline,
        "encodings"
    ]

    docker = subprocess.Popen(docker_cmd)
```

```
    docker.wait()

    shutil.move(tmp_path, output_path + "/" + codec['name'] + " " + parameters + "")
```

measure_variation.py

```
#!/usr/bin/env python3
import json
import os
import shutil
import subprocess

tmp_path = "/tmp/encodings"
out_path = "variations"
cmdline = "-c:v libx264rgb -preset ultrafast -crf 0 -tune stillimage -x264-params keyint=30:scenecut=0:keyint_min=30"
    + " -f mp4 -movflags +frag_keyframe+empty_moov+default_base_moof /out/video.mp4"

sample_sizes = [1, 2, 5, 10, 15, 20, 30]

if not os.path.exists(out_path):
    os.makedirs(out_path)

for sample_size in sample_sizes:
    print(sample_size)

    if not os.path.exists(tmp_path):
        os.makedirs(tmp_path)

    docker_cmd = [
        "docker", "run",
        "--rm", "-it",
        "--name", "encodings",
        "--shm-size", "2g",
        "-v", tmp_path + ":/out",
        "-e", "CODEC_CMDLINE=" + cmdline,
        "-e", "SAMPLE_SIZE=" + str(sample_size),
        "encodings"
    ]

    docker = subprocess.Popen(docker_cmd)
    docker.wait()

    shutil.move(tmp_path + "/data.json", out_path + "/data" + str(sample_size) + ".json")
```

B.5 Number processing

Below are the two scripts (and shared library) that process the collected data and return a CSV file with the results.

crunch_numbers.py

```
#!/usr/bin/env python3
import json
import os
import csv
import numpy as np

from shared import crunch_number
```

```
directory = os.fsencode("out")
folders = []
for file in os.listdir(directory):
    filename = os.fsdecode(file)
    if filename == ".DS_Store":
        continue
    folders.append(filename)

folders.sort()

keys = ['wall', 'user', 'system', 'read', 'write', 'bitrate', 'avg_memory', 'max_memory']
rows = [['name'] + keys]
for codec in folders:
    data, samples = crunch_number("out/" + codec + "/data.json")
    fields = [codec]
    for key in keys:
        fields.append(round(data[key], 4))
    rows.append(fields)

with open('data/all.csv', 'w') as file:
    writer = csv.writer(file, delimiter=';', quoting=csv.QUOTE_ALL)
    writer.writerows(rows)
```

crunch_variation_numbers.py

```
#!/usr/bin/env python3
import json
import os
import csv
import numpy as np
import scipy.stats
from shared import crunch_number

# https://stackoverflow.com/a/15034143/6397601
def mean_confidence_interval(data, confidence=0.95):
    a = 1.0 * np.array(data)
    n = len(a)
    m, se = np.mean(a), scipy.stats.sem(a)
    h = se * scipy.stats.t.ppf((1 + confidence) / 2., n-1)
    return m, m-h, m+h


directory = os.fsencode("variations")
files = []
for file in os.listdir(directory):
    filename = os.fsdecode(file)
    if filename == ".DS_Store":
        continue
    files.append(filename)

files.sort()

keys = ['wall', 'user', 'system', 'read', 'write', 'bitrate', 'avg_memory', 'max_memory']
rows = [['name'] + keys + ['bitrate gamma', 'bitrate t_u', 'bitrate t_v', 'cpu gamma', 'cpu t_u', 'cpu t_v']]
for file in files:
    data, samples = crunch_number("variations/" + file)

    fields = [file]
    for key in keys:
        fields.append(round(data[key], 4))

    gamma, t_u, t_v = mean_confidence_interval(samples['bitrate'])
    fields += [round(gamma, 4), round(t_u, 4), round(t_v, 4)]
```

```
    cpu_samples = samples['duration']['user'] + samples['duration']['system']
    gamma, t_u, t_v = mean_confidence_interval(cpu_samples)
    fields += [round(gamma, 4), round(t_u, 4), round(t_v, 4)]

    rows.append(fields)

with open('data/variation.csv', 'w') as file:
    writer = csv.writer(file, delimiter=';', quoting=csv.QUOTE_ALL)
    writer.writerows(rows)
```

shared.py

```
#!/usr/bin/env python3
import json
import os
import csv
import numpy as np

def last_not_zero_value(lst):
    for value in reversed(lst):
        if value != 0.0:
            return value
    return 0.0

def crunch_number(file):
    with open(file, "r") as f:
        data = json.load(f)

    bitrates = []
    memory_usages = []
    duration = {
        "wall": [],
        "user": [],
        "system": []
    }
    io = {
        "read": [],
        "write": []
    }

    # Extract the values from the measurements
    for measurement in data['measurements']:
        bitrates.append(measurement['bitrate'])
        memory_usages.append(last_not_zero_value(measurement['memory']))

        for key in duration:
            duration[key].append(last_not_zero_value(measurement['times'][key]))

        for key in io:
            io[key].append(last_not_zero_value(measurement['io'][key]))

    # Calculate the averages for times and io
    output = {}
    for key in duration:
        output[key] = float(np.median(duration[key]))
    for key in io:
        output[key] = float(np.median(io[key]))

    # Accumulate individual values
    output['bitrate'] = float(np.median(bitrates))
    output['avg_memory'] = float(np.median(memory_usages))
    output['max_memory'] = float(np.max(memory_usages))

    samples = {}
```

```
    samples['bitrate'] = bitrates
    samples['memory'] = memory_usages
    samples['duration'] = duration
    samples['io'] = io

    return (output, samples)
```