The Case for Offload Shaping

Wenlu Hu, Brandon Amos, Zhuo Chen, Kiryong Ha, Wolfgang Richter,
Padmanabhan Pillai†, Benjamin Gilbert, Jan Harkes, Mahadev Satyanarayanan

Carnegie Mellon University and †Intel Labs

ABSTRACT

When offloading computation from a mobile device, we show that it can pay to perform additional on-device work in order to reduce the offloading workload. We call this *offload shaping,* and demonstrate its application at many different levels of abstraction using a variety of techniques. We show that offload shaping can produce significant reduction in resource demand, with little loss of application-level fidelity.

1. Introduction

Offloading computation from a mobile device to the cloud or a cloudlet is a well-known technique for improving performance and extending battery life [5, 6, 9, 16]. This includes optimal partitioning of a computational pipeline into early stages that are executed locally, and later stages that are executed remotely. The partitioning may vary dynamically, depending on the supply and demand of resources such as network bandwidth, energy, and cache space [3, 7, 8].

In this paper, we show that it is sometimes valuable to perform *additional* cheap computation, not part of the original pipeline, on the mobile device in order to modify the offloading workload. We call this *offload shaping.* We show that offload shaping can be applied at many different levels of abstraction using a variety of techniques, and that it can produce significant reduction in resource demand with little loss of application-level fidelity or responsiveness.

We begin in Sections 2 and 3 by suppressing transmission of blurry images in video streams. In later sections, we advance to more sophisticated techniques. We conclude by motivating an API through which a cloud service can communicate application-specific offload shaping information to a mobile device.

2. Example: Blurry Video

Object recognition within frames in a live video stream is an example of a computationally expensive task that benefits greatly from offloading [11]. Unfortunately, some video frames may be blurry due to user movement, poor camera focus, moisture on the camera lens, or other reasons. Figure 1 shows two frames of the same scene: image (a) is sharp and image (b) is blurred by camera motion. Blurry images can adversely affect the accuracy of computer vision algorithms.

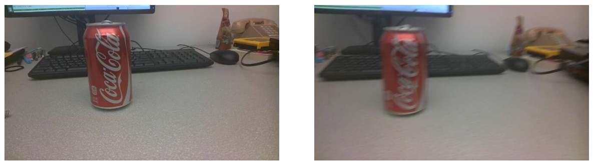

(a) Sharp (b) Blurry

Figure 1: Sharp and blurry frames of same scene

Coke Can	Frames Judged by User 1		Frames Judged by User 2	
	Sharp	Blurry	Sharp	Blurry
Detected	161	5	166	0
Not detected	2	91	4	89

Figure 2: Impact of blurry frames on accuracy

Figure 2 shows the measured accuracy of object recognition (specifically, the red Coke can seen in Figure 1), using the MOPED algorithm for object detection [4]. Each of the 259 frames in the test video, captured using a Google Glass device at 640x360, contains one instance of the object. Hence, an ideal recognizer would find exactly one object in each frame. However, due to head movement during video capture, some of the frames are blurry. On the 163 frames judged to be sharp by User 1, there were only two objects missed by MOPED (false negatives). However, on the 96 frames judged to be blurry by User 1, there were 91 objects missed by MOPED. User 2 produces similar results. Clearly, recognition accuracy suffers when a frame is blurry. We expect other first-person videos taken with head-mounted cameras to also have considerable numbers of blurry frames.

It is a waste of wireless bandwidth and mobile device energy to transmit blurry frames to the offload engine, and to wait for a response that is likely not meaningful. If a mobile

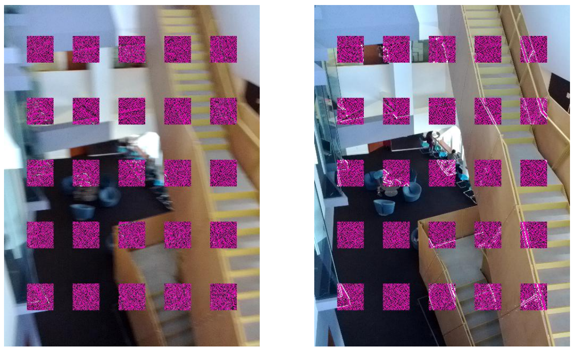
	Send all	Drop blurry
Bytes transferred	0.51M	0.34M
Glass energy (J)	429(2)	292(3)
Server CPU usage (normalized)	1.00(0.01)	0.81(0.01)

Numbers in parentheses are standard deviations from 4 runs. H.264 encoding is used.

Figure 3: Benefits of dropping blurry frames

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

HotMobile'15, February 12–13, 2015, Santa Fe, New Mexico, USA.
Copyright © 2015 ACM 978-1-4503-3391-7/15/02 ...$15.00.
http://dx.doi.org/10.1145/2699343.2699351.


(a) Blurry (b) Sharp

We apply a Sobel filter to 25 patches. Here, the resulting gradients are shown overlaid on the original image.

Figure 4: Blur detection by Sobel filter

		Detected by Sobel Filter	
		Blurry	Sharp
Ground Truth	Blurry	74	22
	Sharp	6	157

Figure 5: Accuracy of Sobel blur detection

device could cheaply and reliably detect blurry frames, it could suppress transmission of those frames with hardly any loss of accuracy. In the example video above, over a third of the frames can be dropped safely at the device, significantly reducing bandwidth and energy consumed. Figure 3 summarizes the measured benefits of dropping the blurry frames at the Google Glass device. In the next section, we explore a number of ways of cheaply detecting blurry frames.

3. Cheap Blur Detection

3.1 Using Image Content

Detecting blur is a well-studied problem in image processing [19]. Intuitively, image gradients will be more gradual when an image is blurry because the edges in the image are less sharp. We use a Sobel operator [18] to compute image intensity gradients. To keep computational costs low on mobile devices, our implementation samples 25 patches distributed over the image. If the gradient at any of the samples exceeds a threshold, we deem the image to be sharp. Only if the gradient is below the threshold in all samples do we deem it to be blurry. Figure 4 shows the results of this technique applied to two similar images that differ in sharpness. We verify this approach on the video used in Figure 1, comparing its results against the ground truth as judged by User 1. Despite its simplicity, this method matches human-judged blurriness with high accuracy, as shown in Figure 5.

We implement an offload-shaping *filter* on Google Glass by using a Sobel operator to drop blurry frames before transmission to a remote MOPED object detection service. We measure the effects on energy, bandwidth, and latency. In these experiments, frames transmitted from Glass are encoded in H.264 with B-frames disabled to satisfy low latency requirements of real-time applications. They are sent to a remote MOPED object detection service. This service runs on a *cloudlet* [17], a server-class machine running a cloud software stack and connected to the same LAN as the WiFi base station. We use a dedicated WiFi access point to reduce interference and maximize bandwidth, thus favoring offload without shaping. To stabilize Glass performance, ice packs are used to cool the device externally [10], and a Bluetooth connection is established between Glass and a phone but not used for data transmission [2]. The Glass screen is kept off as it is not useful for this type of application. We repeat each experiment 4 times, and report both mean and standard deviations (in parentheses). As MOPED is nondeterministic, accuracy results have some variability even on the same input. Unless noted otherwise, all experiments in this paper use this experimental setup.

	No shaping	Drop blurry	Improvement
Bytes transferred	0.51M	0.37M	27%
Frames recognized	171(2)	162(1)	-5%
E2E latency (ms)	920(8)	859(14)	7%
Glass power (W)	1.82(0.01)	1.82(0.02)	0%
Glass energy (J/frame)	1.66(0.01)	1.51(0.02)	9%
Server CPU usage (normalized)	1.00(0.01)	0.84(0.02)	16%

Figure 6: Blur detection with Sobel filter

Figure 6 shows that dropping blurry frames results in significant reductions in the bytes transferred (27%) and processing cycles used on the server (16%), as well as a modest reduction in the average end-to-end processing latency per frame (7%). The mean latency improves because results for dropped frames are known quickly. Because frames are processed sequentially after the results of the prior ones are returned, this also results in an increase in the processing rate. So although Glass power is unaffected, the average energy consumed per frame improves by 9%. All of these improvements come at a slight 5% reduction in the object detection accuracy. We note that even with "perfect" blurry-frame dropping (i.e., using User 1's labels from Figure 2), we would have a similar 5% reduction in MOPED accuracy.

3.2 Using On-Board Sensors

Mobile devices such as smartphones and Google Glass devices have on-board sensors such as accelerometers and gyroscopes. These sensors are attached to the same rigid object as the camera, so their readings are correlated with camera motion and any resulting image quality degradation.

At first glance, the accelerometer seems to be an obvious sensor for detecting blur. However, our experiments show otherwise. This is because blur is correlated with camera velocity, but the accelerometer measures the derivative of velocity (i.e., acceleration). In movements such as shaking one's head when wearing Google Glass, acceleration tends to peak when linear velocity is low. Integrating acceleration readings to yield velocity does not work well either, because large errors quickly accumulate.

Fortunately, gyroscopes are increasingly common in mobile devices and turn out to be effective at predicting blur. Gyroscopes emit angular velocity in $radians/s$. Figure 7 shows a strong correlation between gyroscope readings and blurriness ground truth, i.e. User 1's label in our test video from Figures 1 and 2.

We conclude a frame is blurry when its corresponding gyroscope reading exceeds a threshold. Figure 8 shows the trade-off between frames dropped and accuracy as this threshold is changed. Curve (a) is based on the video used above,

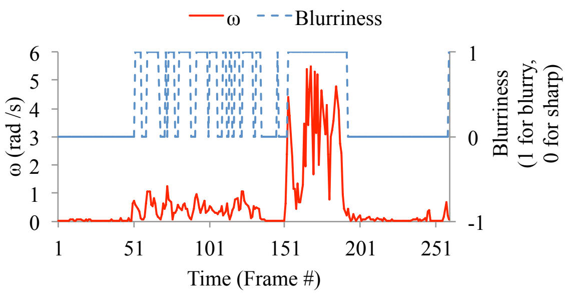


Figure 7: Correlation between blurriness and gyroscope readings ω

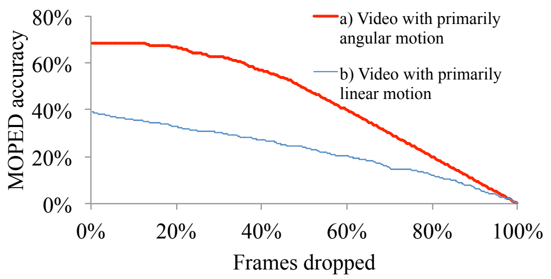


Figure 8: Frames dropped vs. MOPED accuracy while gyroscope threshold changes

where blurriness is caused by head movement. In this case, motion is primarily angular and readily detected by the gyroscope. Here, the strategy is effective at selecting the right frames to drop, so many can be dropped before significantly affecting accuracy. We also test on a second video primarily containing linear motion. In this curve (b), gyro readings are not helpful in selecting the right frames to drop, so accuracy suffers immediately with any dropped frames. In practice, angular movements have a greater effect on blurriness, since they affect the entire scene, while effects of linear movement drop off rapidly with distance.

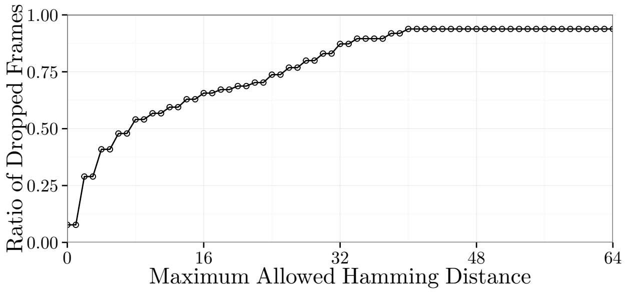
Using the methodology from Section 3.1 and a threshold of $0.5 rad/s$, we test the effectiveness of a filter that uses the gyroscope to predict blurriness. The results in Figure 9 show significant reductions in bytes transferred, average processing latency, and server load. Although power increases slightly on Glass, the improvement in throughput leads to better energy efficiency and the energy per frame improves significantly (20%). Once again, these improvements are achieved with only a small reduction in accuracy (8%).

4. Exploiting Inter-Frame Similarity

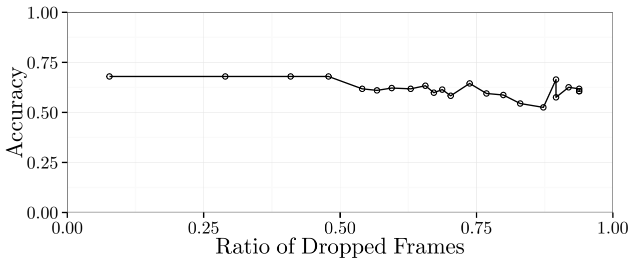
In live video streamed from a mobile device, the scene does not often change appreciably between consecutive frames.

	No shaping	Drop blurry	Improvement
Bytes transferred	0.51M	0.37M	27%
Frames recognized	171(2)	157(1)	-8%
E2E latency (ms)	920(8)	750(3)	18%
Glass power (W)	1.82(0.01)	1.87(0.01)	-3%
Glass energy (J/frame)	1.66(0.01)	1.33(0.01)	20%
Server CPU usage (normalized)	1.00(0.01)	0.84(0.00)	16%

Figure 9: Blur detection with gyroscope filter



(a) Frames Dropped



(b) MOPED Accuracy vs. Dropped Ratio

Figure 10: Effects of similarity filter

The results of many computer vision algorithms will likely remain constant when applied to sequences of nearly identical frames. Hence, sufficiently similar frames can be safely discarded to conserve bandwidth and energy.

How can we determine whether a frame is very similar to the preceding one? This is much harder than it appears at first glance. Directly comparing pixel values of adjacent frames is not effective. Semantically insignificant changes (e.g., minor camera movement, or fluorescent light flicker) can cause a huge number of pixels to differ and result in a large computed distance between frames, while meaningful changes (e.g., objects moved within a scene) may not affect many pixels and will result in a small computed distance.

Fortunately, the problem of image similarity has been well studied in the literature. A family of techniques called *perceptual hashing* has been developed to encode visual properties of images into bit strings [14, 15]. Importantly, distance metrics between these image "hashes" do correspond to perceived similarity between the source images, and are robust to semantically insignificant changes.

We implement a perceptual hashing [20] filter based on a 64-bit discrete cosine transform and a simple Hamming-distance metric to determine similarity between frames. If a frame is sufficiently similar to the last transmitted frame, it is dropped, and the output of downstream processing is assumed to remain constant. We compare a frame to the last transmitted frame and not just to the previous frame; otherwise, sequences of significant but slowly-accumulated changes may be dropped entirely. We also force transmission of a frame after 15 consecutive frames have been dropped. This approach is better than a naive sampling of frames, as it can respond immediately to sudden changes in the scene, without waiting for the next sampling interval.

Figure 10 shows the filter's impact on MOPED accuracy on the video used in Figure 1. On a Google Glass device, the hash and distance metrics can be computed in about 20 ms. Figure 10(a) shows that the hashes in our experi-

	No shaping	Drop similar	Improvement
Bytes transferred	0.51M	0.23M	55%
Frames recognized	171(2)	189(1)	11%
Glass power (W)	1.82(0.01)	1.83(0.01)	-1%
E2E latency (ms)	920(8)	393(2)	57%
Glass energy (J/frame)	1.66(0.01)	0.72(0.01)	57%
Server CPU usage (normalized)	1.00(0.01)	0.27(0.01)	73%

Figure 11: Similarity filter performance

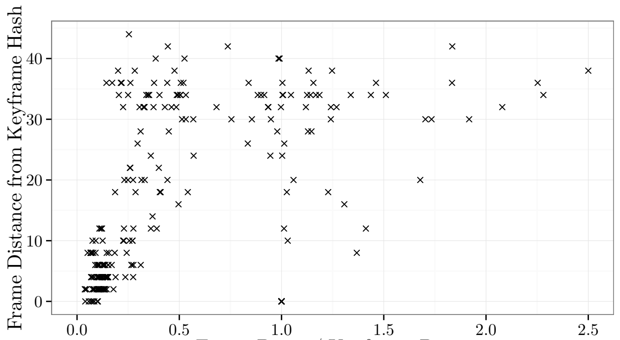
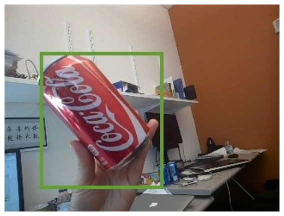


Figure 12: Similarity vs. H.264 encoded frame size (normalized)

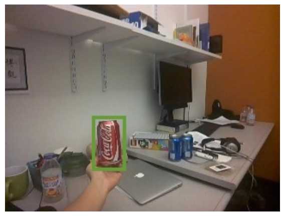
ment never differ by more than 40 bits with the keyframe hash. Figure 10(b) shows the trade-off between dropping more frames and the MOPED accuracy. Unlike in the previous sections, where dropped frames are interpreted as detection failures, we report the detector output from the last transmitted frame for dropped frames. As a result, the relationship between accuracy and dropped frames is complex, and can vary depending on the subset of frames sent.

Figure 11 shows the reduction in energy and bytes transferred with the test video. We are sending fewer than half the bytes. As explained in Section 3.1 and 3.2, even though Glass power increases by 1%, as frames are processed at twice the original rate, energy consumed for each frame on Glass is less than half its original value. The server CPU usage also drops to almost one fourth its original value. Here, we again see significant reduction in cost, with little difference in fidelity.

Since video encoding algorithms are based on encoding differences between frames, can we use the size of an encoded frame to estimate its similarity to the preceding one? With H.264 encoding, we did not find a clear relationship with the size of the compressed frame, but did find a statistical correlation between similarity and data size when normalized to the preceding keyframe size (Figure 12). Here, the encoding was based on GOP (interval between keyframes) of 10, used an x264 "medium" preset with a "zero-latency" tuning option, and omitted B-frames. The correlation is very noisy, and whether normalized encoded frame size is useful for predicting similarity is left for future research. Furthermore, given the vast literature on video indexing and key frame selection [12], there may be other encoding techniques that provide a better correlation with similarity that can be leveraged for offload shaping.



(a) Video with a large Coke can (446 frames)



(b) Video with a normal-size Coke can (510 frames)

Figure 13: Example frames for red filter

	No shaping	Send red only	Improvement
Bytes transferred	8.6M	2.8M	67%
Frames recognized	396(3)	380(5)	-4%
E2E latency (ms)	471(12)	153(2)	68%
Glass power (W)	1.80(0.01)	1.99(0.02)	-11%
Glass energy (J/frame)	0.84(0.01)	0.28(0.01)	67%

MJPEG encoding is used to deal with varying frame size.

Figure 14: Red filter with MOPED server

5. Context-sensitive Offload Shaping

Blurriness and inter-frame similarity are broadly applicable mechanisms for offload shaping, regardless of the application context. In this section, we show that strategies tailored specifically to particular application contexts can also reduce mobile resource usage. For example, every image that contains the Coke can shown in Figure 1 will have a patch of red in it. Hence, any frame lacking a red patch cannot contain a Coke can. Discarding such frames achieves effective offload shaping specific to the context of Coke-can detection. However, this specialized filter will not be useful in other contexts, such as finding blue cars.

Depending on the context, in addition to dropping unnecessary frames, it may be possible to crop the useful frames to remove unneeded background. For example, for a face recognition application, only faces are interesting. If we can use a low-cost method to detect and crop the faces on the mobile device, most of the pixels do not need to be transmitted to the back-end recognition service.

5.1 Example: Color Filter

We first use color to perform offload shaping for Coke-can detection. To select useful patches, we implement a simple red color filter using Android OpenCV. We convert frames to the HSV color space and use a simple distance threshold to find pixels close to the desired color. We crop a rectangular region that encloses the largest connected component of the desired color, with a narrow, fixed-width margin. Only this cropped region is sent to the server.

We use a mostly-sharp video, shown in Figure 13(a), to test the color filter. The green box shows the region that is cropped out and sent to the server. Results in Figure 14 show significant reduction (around two thirds) in bytes transferred and average latency. As explained in Sections 3.1 and 3.2, although the Glass power increases slightly, the energy consumed per frame drops to one third of the original. All of these benefits are achieved with only a small reduction in MOPED accuracy (4%). The benefits in this case are limited by the large size of the Coke can in the frames. For the video in Figure 13(b), where Coke can sizes are smaller

Video description	Whole frames (MB)	Faces only (MB)	Savings
One face, still	53.8	3.7	93%
Two faces, still	65.0	6.0	91%
Two faces, moving	65.8	4.9	92%

Each video is around one minute.

Figure 15: Face detection on Bosch cameras

and more realistic, only one tenth of the bytes are sent after applying the red filter and one fourth of time is needed to process each frame. Note that the video resolution here is 360x240. Our filters will make an even larger difference for videos with higher resolutions, as they make greater demands on network bandwidth.

5.2 Leveraging Hardware Accelerators

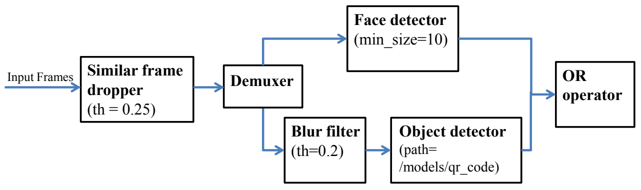
Many cameras and devices such as Google Glass now include features such as hardware accelerated face detection. Face detection in hardware helps consumer-grade cameras efficiently focus on what is commonly the most important part of any frame. Many expensive cameras and an increasing number of smartphones now also stabilize images actively. They move the lens in real-time to counter the effect of camera movement. This can be viewed as hardware accelerated blur correction. In addition, a wide range of surveillance cameras have on-board computer vision programs to generate abstract information about surveilled scenes. These programs can be re-purposed for offload shaping. For common tasks, hardware acceleration will be both quicker and more efficient than software targeting a general-purpose processor. Offload shaping can therefore leverage hardware-based as well as software-based filtering.

To explore this possibility we experimented with a Bosch surveillance camera, the Dinion HD 1080p HDR. For applications that are only interested in human faces, such as mood or face recognition, this camera's cheap, built-in face detection functionality is helpful. We built a face filter, based on the Dinion's on-board face detection, that simply crops and transmits only the faces in each frame. Figure 15 shows significant savings in bytes transferred.

6. Offload Shaping API

We have shown that offload shaping can help improve bandwidth use, response time, and energy efficiency of offloaded tasks. In addition, the best strategies for offload shaping are often context- and application-specific. In order to handle the diverse needs of different applications, the mobile device should have a simple API for dynamic specification of offload shaping. With this API, the applications can dynamically specify and adjust the shaping they want according to the context. Although we have been focusing on individual shaping techniques so far, it would also be useful to combine multiple techniques. Thus, the API should support composing components. Finally, we would like the system to be extensible, so new offload-shaping filters can easily be added to the system.

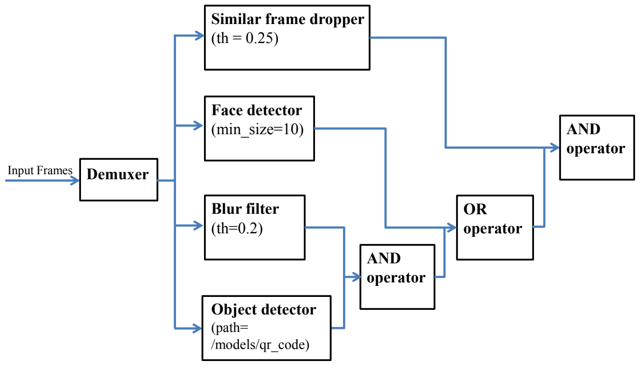
Rather than design such an API from scratch, we observe that many existing software libraries and frameworks have similar requirements and have already-established APIs to satisfy them. In particular, the GStreamer API [1] seems to fit our needs well. It was designed as a pluggable system to compose multimedia transcoding pipelines, but can be easily



Input Frames
Similar frame dropper (th = 0.25)
Demuxer
Face detector (min_size=10)
Blur filter (th=0.2)
Object detector (path= /models/qr_code)
OR operator

```
gst-launch input ! similar_frame_dropper th=0.25 ! demuxer name=dmx ! face_detector min_size=10
                   ! or_operator name=or
              dmx. ! blur_filter th=0.2 ! object_detector path=/models/qr_code ! or.
```

(a) Relatively sequential



Input Frames
Demuxer
Similar frame dropper (th = 0.25)
Face detector (min_size=10)
Blur filter (th=0.2)
Object detector (path= /models/qr_code)
AND operator
OR operator
AND operator

```
gst-launch input ! demuxer name=dmx ! similar_frame_dropper th=0.25 ! and_operator name=and2
              dmx. ! face_detector min_size=10 ! or_operator name=or ! and2.
              dmx. ! blur_filter th=0.2 ! and_operator name=and1 ! or.
              dmx. ! object_detector path=/models/qr_code ! and1.
```

(b) Fully parallel

Figure 16: Example of declarative APIs and corresponding diagrams

adapted for specifying complex offload shaping policies. In addition to a programmatic interface, it has a declarative, text-based configuration language that lets us launch a set of components, express a graph of how they connect, and provide configuration strings for individual components if desired. Finally, the GStreamer framework is supported on all major operating systems, including Android and iOS.

Figure 16(a) shows an example of a complex offload-shaping strategy expressed with the GStreamer framework. Here, frames are first filtered based on similarity, and of those that pass, only frames with faces or non-blurry QR codes are ultimately transmitted from the device. To accomplish this, we need to create a library of new GStreamer components that implement the individual offload-shaping algorithms. We also need to implement logical AND / OR filters, as GStreamer has no such concepts. In addition, to keep the branches of a pipeline in sync, we replace dropped frames with small placeholders. The example also shows how components can be configured, using simple short strings. For more complex configuration, such as object models, we rely on configuring with paths or URLs to the needed data.

This declarative API is also quite flexible. Figure 16(b) shows how to express the same policy in a more parallel fashion. On a mobile device that has multiple cores or parallel sets of hardware accelerators, this version could be executed with lower latency, as the main tasks are run in parallel. Of course, this results in additional processing for frames that would have been dropped early in the sequential pipeline. The trade-off between response time and mobile resource use can be tailored to each specific application.

The GStreamer API helps us compose multiple filters in a configurable way. Such filters might be thin wrappers around existing libraries or more complicated context-specific filters implemented as custom code. Applications can distribute these filters as dynamic libraries, and could upload them to a central archive for downloading, much as multimedia CODECs are handled today. We also imagine that the most commonly-used filters will be pre-installed on mobile devices in the future and ready to use for offload shaping.

7. Conclusion

Offload shaping demonstrates that judicious use of additional computation on a mobile device can significantly improve resource usage in an offload system. Offload shaping improves cloud offloading by combining the use of hints to speed up computations with the concept of early discard [13]. The on-board sensors and computing power of mobile devices make it possible to achieve accurate early discard. In this paper, we explore various approaches to dropping low-value input data without sacrificing application fidelity. Such approaches can save significant resources on mobile devices, including processing time, network bandwidth, and energy. They also improve scalability of the backend server, reducing inbound traffic and workload.

Offload shaping is generally valuable to a wide range of applications. Our proposed API allows individual applications to tailor a shaping strategy to their specific needs. We will explore how various applications benefit from this API in future work. This API is extensible, allowing applications to incorporate new sources of hints such as GPS coordinates, barometric pressure readings, and camera metadata. As mobile platforms become more capable, with additional sensors and hardware-accelerated processing, offload shaping will become an increasingly powerful tool enabling energy-efficient cloud and cloudlet offload.

Acknowledgements

We wish to thank Rahul Sukthankar and Rajen Bhatt for their insights and guidance on various aspects of this research. This research was supported by the National Science Foundation (NSF) under grant number IIS-1065336, by an Intel Science and Technology Center grant, by DARPA Contract No. FA8650-11-C-7190, and by the Department of Defense (DoD) under Contract No. FA8721-05-C-0003 for the operation of the Software Engineering Institute (SEI), a federally funded research and development center. This material has been approved for public release and unlimited distribution (DM-0000276). Additional support was provided by IBM, Google, Bosch, Vodafone, and the Conklin Kistler family fund. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and should not be attributed to their employers or funding sources.

8. REFERENCES

[1] GStreamer: open source multimedia framework. `http://gstreamer.freedesktop.org/`, 2014.

[2] Issue 512: Bluetooth connection interfering with TCP Traffic on WiFi. `https://code.google.com/p/google-glass-api/issues/detail?id=512`, May 2014.

[3] R. Balan, M. Satyanarayanan, T. Okoshi, and S. Park. Tactics-based Remote Execution for Mobile Computing. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*, San Francisco, CA, May 2003.

[4] A. Collet, M. Martinez, and S. S. Srinivasa. The MOPED framework: Object Recognition and Pose Estimation for Manipulation. *The International Journal of Robotics Research*, 2011.

[5] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: Making Smartphones Last Longer with Code Offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, San Francisco, CA, June 2010.

[6] J. Flinn. *Cyber Foraging: Bridging Mobile and Cloud Computing via Opportunistic Offload*. Morgan & Claypool Publishers, 2012.

[7] J. Flinn, D. Narayanan, and M. Satyanarayanan. Self-Tuned Remote Execution for Pervasive Computing. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems*, Schloss Elmau, Germany, May 2001.

[8] J. Flinn, S. Park, and M. Satyanarayanan. Balancing Performance, Energy Conservation and Application Quality in Pervasive Computing. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, Vienna, Austria, July 2002.

[9] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen. COMET: Code Offload by Migrating Execution Transparently. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, Hollywood, CA, October 2012.

[10] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan. Towards wearable cognitive assistance. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, Bretton Woods, NH, June 2014.

[11] K. Ha, P. Pillai, G. Lewis, S. Simanta, S. Clinch, N. Davies, and M. Satyanarayanan. The Impact of Mobile Multimedia Applications on Data Center Consolidation. In *Proceedings of the IEEE International Conference on Cloud Engineering*, San Francisco, CA, March 2013.

[12] W. Hu, N. Xie, L. Li, X. Zeng, and S. Maybank. A survey on visual content-based video indexing and retrieval. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 41(6):797–819, 2011.

[13] L. Huston, R. Sukthankar, R. Wickremesinghe, M. Satyanarayanan, G. R. Ganger, E. Riedel, and A. Ailamaki. Diamond: A storage architecture for early discard in interactive search. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, San Francisco, CA, March 2004.

[14] S. S. Kozat, R. Venkatesan, and M. K. Mihçak. Robust perceptual image hashing via matrix invariants. In *Image Processing, 2004. ICIP'04. 2004 International Conference on*, volume 5, pages 3443–3446. IEEE, 2004.

[15] V. Monga and B. L. Evans. Perceptual image hashing via feature points: performance evaluation and tradeoffs. *Image Processing, IEEE Transactions on*, 15(11):3452–3465, 2006.

[16] M. Satyanarayanan. Pervasive Computing: Vision and Challenges. *IEEE Personal Communications*, 8(4), 2001.

[17] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The Case for VM-Based Cloudlets in Mobile Computing. *IEEE Pervasive Computing*, 8(4):3 (Sidebar: "Help for the Mentally Challenged"), October-December 2009.

[18] I. Sobel and G. Feldman. A 3x3 isotropic gradient operator for image processing. A talk at the Stanford Artificial Intelligence Project, 1968.

[19] R. Szeliski. *Computer Vision: Algorithms and Applications*. Springer, 2010.

[20] C. Zauner. *Implementation and benchmarking of perceptual image hash functions*. PhD thesis, University of Applied Sciences Hagenberg, Austria, 2010.