Characterizing the Reconfiguration Latency of Image Sensor Resolution on Android Devices

Jinhan Hu, Jianan Yang, Vraj Delhivala, Robert LiKamWa
Arizona State University
Tempe, Arizona
jinhanhu@asu.edu,jyang168@asu.edu,vdelhiva@asu.edu,likamwa@asu.edu

ABSTRACT

Advances in vision processing have ignited a proliferation of mobile vision applications, including augmented reality. However, limited by the inability to rapidly reconfigure sensor operation for performance-efficiency tradeoffs, high power consumption causes vision applications to drain the device's battery. To explore the potential impact of enabling rapid reconfiguration, we use a case study around marker-based pose estimation to understand the relationship between image frame resolution, task accuracy, and energy efficiency. Our case study motivates that to balance energy efficiency and task accuracy, the application needs to dynamically and frequently reconfigure sensor resolution.

To explore the latency bottlenecks to sensor resolution reconfiguration, we define and profile the end-to-end reconfiguration latency and frame-to-frame latency of changing capture resolution on a Google LG Nexus 5X device. We identify three major sources of sensor resolution reconfiguration latency in current Android systems: (i) sequential configuration patterns, (ii) expensive system calls, and (iii) imaging pipeline delay. Based on our intuitions, we propose a redesign of the Android camera system to mitigate the sources of latency. Enabling smooth transitions between sensor configurations will unlock new classes of adaptive-resolution vision applications.

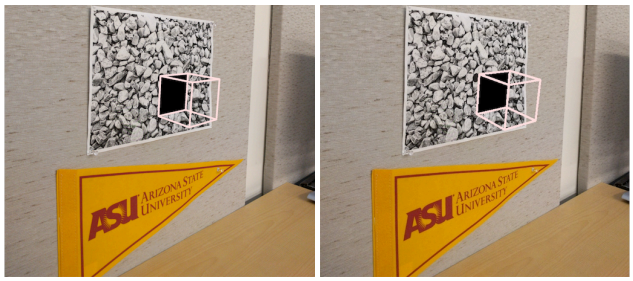
KEYWORDS

Image sensor; Camera System; Reconfiguration; Mobile devices; Operating system optimization

ACM Reference format:
Jinhan Hu, Jianan Yang, Vraj Delhivala, Robert LiKamWa. 2018. Characterizing the Reconfiguration Latency of Image Sensor Resolution on Android Devices. In *Proceedings of 19th International Workshop on Mobile Computing Systems and Applications, Tempe , AZ, USA, February 12–13, 2018 (HotMobile '18),* 6 pages.
https://doi.org/10.1145/3177102.3177109

1 INTRODUCTION

The accuracy and performance of vision processing on mobile devices promises a proliferation of vision-powered possibilities. For example, real-time face recognition can identify faces to track interpersonal interactions [2], object recognition can observe road signs for navigation [4], and augmented reality (AR) can integrate virtual objects with the physical world for immersive user experiences. Combined with a wide adoption of strong computational and sensing hardware on mobile devices, the trend of algorithmic improvements for computer vision has led companies to issue broad development support through computer vision frameworks, e.g., Google Mobile Vision, OpenCV and AR frameworks, e.g., Apple ARKit, Google ARCore, PTC Vuforia.



(a) At 1860x2480 resolution, estimated pose has L2-norm translation error 0.87 cm, rotation error 0.82°.

(b) At 720x960 resolution, estimated pose has L2-norm translation error 1.15 cm, rotation error 8.51°.

Figure 1: Low resolution frames cause slight errors in camera pose estimation, creating visible gaps in geometry.

However, the energy efficiency of vision applications is severely limited by the inability to rapidly configure sensor operation. Previous research has focused on improving the energy-efficiency of processing image data, such as bypassing traditional image signal processing stages [3], pushing neural network processing into the analog domain of the image sensor [11], and offloading sensor processing to the cloud [15]. However, it is well understood that the power consumption of *capturing* sensor data dominates system power consumption (the SONY IMX377 image sensor draws over 39% of Nexus 5X system power consumption [5]) and can vary significantly with spatiotemporal resolution [12].

Indeed, enabling an ability to tune a sensor's spatial resolution to downsample a field-of-view or focus on regions-of-interest would allow a system to balance energy efficiency and image fidelity for visual tasks [11]. For example, an application could briefly capture high resolution image frames to locate distant objects with high precision, while capturing low-resolution frames to display nearby information to the user with high energy efficiency. Unfortunately, the act of reconfiguring sensor operation for different resolutions

HotMobile '18, February 12–13, 2018, Tempe , AZ, USA

ACM ISBN 978-1-4503-5630-5/18/02…$15.00
https://doi.org/10.1145/3177102.3177109

is accompanied by long latency, dramatically dropping application performance as the operating system and sensor hardware coordinate to reconfigure sensor operation.

We define two types of sensor reconfiguration latency that detract from vision application performance in distinct ways. **End-to-end reconfiguration latency** is the time between an application's request to change configuration and the time the application receives a full frame of the new configuration. An increase in end-to-end latency creates an inflexibility to shift between modes of operation. An average end-to-end latency of 400 ms will prevent applications from smoothly transitioning the balance of energy efficiency and image fidelity. However, after the application issues a configuration request, frames of the prior configuration will continue to arrive at the application. Thus, we use **frame-to-frame latency** to indicate the interval between two frames being provided to the application (the latter frame being newly configured). An increase in frame-to-frame latency creates a perception that the system is "dropping frames". Compared to the typical frame-to-frame latency of 33 ms at 30 frames per second (FPS), an average frame-to-frame latency of 267 ms is equivalent to the system dropping 8 camera frames.

In this paper, we explore sources of sensor resolution reconfiguration latency in the Android system, comprehensively profiling a Google LG Nexus 5X. Our measurements indicate an average end-to-end reconfiguration latency of 400 ms and an average frame-to-frame latency of 267 ms. We identify OS bottlenecks in the software stack and sensor pipeline bottlenecks in the hardware system architecture. We observe three major sources of frame-to-frame latency in the Android system: (i) sequential configuration patterns, (ii) expensive system calls, and (iii) imaging pipeline delay.

Built on our understanding, we propose mechanisms to redesign mobile camera systems. Invoking principles of reusing allocated resources, parallelizing independent operations, and timely control of channel management, these mechanisms aim to provide responsive reconfigurability through minimizing end-to-end reconfiguration latency and high-quality user experience during reconfiguration by minimizing frame-to-frame latency.

Thus, in this paper, we contribute the following:

- We use an augmented reality case study to discuss the need for dynamic resolution reconfiguration.
- We profile sources of sensor reconfiguration latency to identify software and hardware bottlenecks.
- We propose mechanisms to redesign low-level operating systems to minimize sensor reconfiguration latency.

2 BACKGROUND

Reconfiguration in the operating system Camera resolution reconfiguration involves all layers of the Android system stack, including the application, the camera framework, the camera Hardware Abstraction Layer (HAL), and the kernel camera device drivers.

The procedure is as follows:

(1) The application sends a resolution request and creates a new camera capture session.
(2) The framework waits for the prior session to stop and calls the HAL to configure streams.
(3) The HAL deletes previous channels and streams.
(4) The framework issues a capture request to the HAL.
(5) The HAL initializes channels and streams, configuring sensor hardware with the capture settings of the output buffer.
(6) The HAL sends captured frames through pipeline stages for image signal processing. The HAL sends fully-processed frames to the framework.
(7) The framework delivers the frame to the application surface and calls a developer-defined callback function.

Reconfiguration in sensor hardware To better understand the reconfiguration from a sensor hardware perspective, we study datasheets provided by image sensor manufacturers [13][14]. In all cases, we find that at the sensor hardware level, the end-to-end hardware reconfiguration latency is one to two frame times after the register reconfiguration request is issued. For a frame time of 33 ms, corresponding to a frame rate of 30 FPS, this can be up to 66 ms. The sensor uses the following sequence of operations to reconfigure:

(1) When the sensor receives a request, the sensor waits for the current frame to capture. Depending on when the request arrives, this can be immediate, or take up to one frame time.
(2) While frame A is being read out, the sensor captures a new frame with the new configuration. This takes one frame time.
(3) The sensor reads out frame B while the next frame is being captured. This takes one frame time to complete.

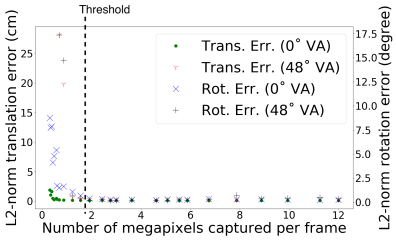
3 CASE STUDY: ADAPTIVE RESOLUTION FOR POSE ESTIMATION

To examine the potential effectiveness of resolution reconfiguration, we present a case study around marker-based pose estimation to understand the relationship between frame resolution, task accuracy, and energy efficiency. Marker-based pose estimation visually locates a physical marker to geometrically register a physical environment with a virtual camera, allowing insertion of virtual objects alongside real environments for AR. User experience is sensitive to the precision of pose estimation; as shown in Fig. 1b, slight errors cause visible gaps in geometry, breaking user immersion. When streaming frames, errors become especially obvious, as the irregular variations cause a feeling of jitter.

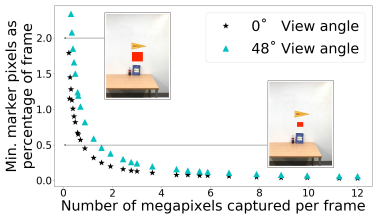
Fortunately, by operating on frames with sufficiently high resolution, pose estimation can minimize errors. Unfortunately, capturing at high resolution draws substantial power, reducing the battery life of mobile devices. Indeed, per-frame system energy consumption scales with resolution, as charted in Fig. 2c. Appropriate frame resolutions will strike a balance between pose estimation accuracy and energy efficiency.

To study this tradeoff, we implement OpenCV's marker-based pose estimation tutorial [17], which extracts visual features from a frame, matches them with marker features using Flann-based matching, and estimates camera pose through a Perspective-n-Point algorithm[1]. This reports the rotation and translation vectors of the camera against a physical marker.

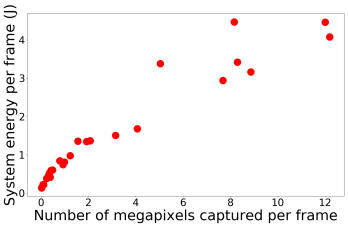
[1]We use SIFT features to optimize for accuracy. Access code and results at https://github.com/hujinhan12/MeteorStudioReconfigurationLatencyHotMobile



(a) At fixed marker distance of 2 m, reducing pixels per frame and increasing viewing angle (VA) will increase L2-norm translation/rotation error.



(b) For error below threshold, reducing pixels per frame and increasing VA requires the marker (in red) to occupy a larger percentage of FOV.



(c) Reducing pixels per frame will reduce system energy consumption for capturing and processing each frame.

Figure 2: Relationship between frame resolution, viewing angle, accuracy, minimum marker pixels, and energy efficiency

Our case study uses a Google Nexus 5X to capture 12 Mp images at different angles and distances from a 7.5 in. by 10 in. marker. We use bilinear downsampling to simulate low-resolution captures from the same pose. This allows us to quantify estimation precision by comparing the estimated rotation and translation vectors of downsampled images against "ground truth" vectors processed from high resolution images.

3.1 Number of marker pixels influences pose estimation accuracy

We find that pose estimation accuracy is dependent upon the percentage of field-of-view (FOV) occupied by marker pixels. As shown in Fig. 2a, at a fixed distance of 2 m, with the camera pointed at the marker, where the marker occupies approximately 1.3% of the field-of-view, the L2-norm estimation error sharply increases as frame resolution is lowered to 480x640, after which the marker cannot be located. This worsens when the camera views the marker at an angle; at a 48° viewing angle, the estimation error sharply increases as resolution is lowered at and below 810x1080. The marker cannot be located in a frame below 600x800 resolution.

Thus, to keep pose estimation error below a threshold, the frame resolution must be large enough that the marker occupies enough pixels for the algorithm to extract features. For small and/or distant markers, high resolution is required, whereas for larger and/or closer markers, lower resolution may suffice. Fig. 2b charts necessary marker coverage as a percentage of the field-of-view for different image resolutions and different viewing angles. As a consequence, depending on (i) the physical distance between the camera and the marker, (ii) the viewing angle between the camera and the marker, and (iii) the size of the marker, the system needs a sufficient frame resolution to precisely locate markers.

3.2 Frame resolution should flexibly adapt to user movement

While energy efficiency motivates the use of minimal resolutions, the implications of Section 3.1 denote a lower limit of acceptable resolution for estimation, given the distance and angle from a marker. This limit of acceptable resolution will change – users will continuously move, changing the camera position with respect to the marker. Moreover, without knowing the location of markers in an environment, the pose estimation application will need to occasionally obtain high resolution frames to locate markers in a scene before using lower resolution captures that suffice to continually estimate the pose of the camera with respect to visible markers.

Thus, to balance energy efficiency and task accuracy, *marker-based pose estimation applications need the ability to dynamically reconfigure sensor operation.* Of course, marker-based pose estimation is just one class of many that would benefit from dynamic sensor reconfiguration; markerless augmented reality, object recognition, and face recognition could all use resolution-based tradeoffs between efficiency and accuracy.

Unfortunately, dynamic reconfiguration is not currently possible without a drop in performance – changing resolution results in hundreds of milliseconds of latency, during which no new frames are provided to the application. In this paper, our goal is to characterize sources of this latency in order to propose mechanisms to overcome the latency.

4 CHARACTERIZE RECONFIGURATION LATENCY IN THE OPERATING SYSTEM

To better understand the sensor resolution reconfiguration latency, we perform an in-depth characterization of the camera library, HAL, and service of the Android OS. In this section, we measure the influence of various configuration functions upon end-to-end and frame-to-frame latency.

Measurement methodology We profile reconfiguration latency on a Google LG Nexus 5X, with a rear-facing 12Mp Sony IMX377 image sensor, loaded with Android Open Source Project, v. 7.1.2. On this device, we instrument an Android test application to request a frame resolution configuration when an on-screen button is clicked. The application cycles through a set of resolutions from 3024x4032 to 120x160.

We inject timing code into Android's camera library framework layer and the camera HAL. Our code prints system timestamps at the beginning and end of each library function and HAL function. These timestamps track the resolution configuration workflow across the Android system stack.

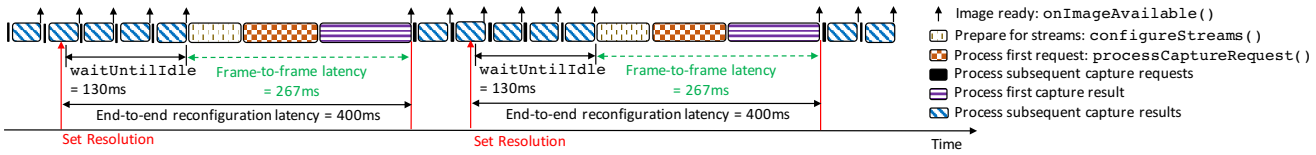


Figure 3: Current sequential configuration patterns incur significant end-to-end and frame-to-frame latency. Shortly after a configuration request, the application continues to receive frames of the previous configuration.

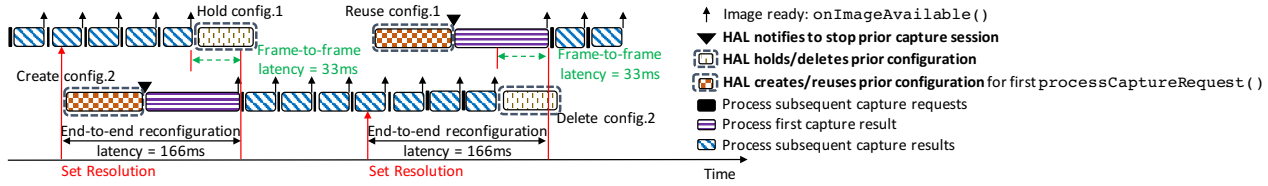


Figure 4: Our proposed alternative introduces parallelizing configuration procedures, reusing configurations, and timely HAL control of channel management, that will reduce the end-to-end and frame-to-frame latency.

Function	Layer	Latency	E-t-E	F-t-F
`waitUntilIdle()`	Library	130 ms	33%	0%
`configureStreams()`	Library	60 ms	15%	22%
`processCaptureRequest()`	Library	84 ms	21%	31%
`process_capture_result()`	HAL	102 ms	26%	38%

Table 1: Latency breakdown, including proportion of end-to-end (E-t-E) and frame-to-frame (F-t-F) latency

4.1 Measuring reconfiguration latency

The operating system implements sensor reconfiguration through interactions between the application camera library, service, and HAL layers. To instrument interactions, we breakdown reconfiguration into critical functional stages, as shown in Table 1. In doing so, we find latency culprits: `waitUntilIdle()`, `configureStreams()`, `processCaptureRequest()`, and `process_capture_result()`.

4.1.1 waitUntilIdle(). After preparing buffers for the new configuration, `waitUntilIdle` forces the process to wait until the system clears the pending request queue from the previous configuration. This `waitUntilIdle` latency forces the HAL to process one stream at a time, only configuring new streams after completing and deleting previous streams.

`waitUntilIdle` latency varies, depending on the number of pending requests in the queue. According to our measurements, end-to-end `waitUntilIdle` latency averages around 130 ms, which contributes almost 33% of the end-to-end reconfiguration latency, as shown in Table 1. Though this end-to-end latency is large, it will not affect frame-to-frame performance since the system still returns frames to the application during idle time, as shown in Fig. 3.

4.1.2 configureStreams(). After clearing the pending request queue for the previous configuration, the framework library invokes `configureStreams` to create the new capture session for the new configuration. Upon invocation, the HAL will stop and release previous channels, streams, and buffers. Subsequently, the HAL will reallocate buffers, reconfigure new streams, and add them to new channels. This sequence involves several system calls. When turning off streams, the HAL invokes 3 `VIDIOC_STREAMOFF ioctl` calls to disable hardware. These `ioctl` calls consume around 20 ms. Adding to the latency, using 18 IPC sockets to release buffers uses 10 ms.

This procedure for deleting old configurations and issuing new configurations occurs when creating any new capture session. The average amount of end-to-end latency for `configureStreams` is around 60 ms, which contributes to 15% of the end-to-end reconfiguration latency. As this operation takes place after previous frames have stopped streaming to the application, `configureStreams` also contributes 22% to frame-to-frame latency.

4.1.3 processCaptureRequest(). After the HAL creates the capture session, the framework layer sends repeated capture requests to the HAL, using `processCaptureRequest` to form each request. As discussed in the background, the HAL must read capture settings and buffer addresses and activate all channels and streams. This invokes several system calls, including IPC socket communication that consumes 14 ms in total, one SensorService JNI library call for sensor hardware configuration that takes 25 ms, and one updating stream information call that consumes 10 ms.

Altogether, the latency of `processCaptureRequest` is around 84 ms, contributing 31% to frame-to-frame latency and 21% to end-to-end reconfiguration latency. Notably, after the first capture request for a given configuration setting, calls to `processCaptureRequest` do not need to initialize channels and streams, and only use 2 ms latency.

4.1.4 process_capture_result(). As the HAL directs frame capture and processing, it uses `process_capture_result` to communicate capture progress to the library framework. A captured frame undergoes several transformation stages in a processing pipeline that can include per-channel white balancing, Bayer pattern demosaicing, and RGB transformations [8]. `process_capture_result`

indicates the remaining pipeline stages for a frame. After a frame leaves the pipeline, the HAL returns the result to the framework.

In steady state, the pipeline returns results every 33 ms. However, the initial time it takes to fill the pipeline creates latency between the capture request and the first result returned. This latency amounts to around 102 ms, which contributes 38% to the frame-to-frame latency and the 26% to the end-to-end reconfiguration latency.

Along with other miscellaneous reconfiguration latencies, amounting to 24 ms, these reconfiguration stages incur 400 ms of end-to-end latency and 267 ms of frame-to-frame latency.

5 IDENTIFY SOURCES OF LATENCY IN THE OPERATING SYSTEM

A deeper look into our characterization exposes three critical sources of latency: *(i)* sequential configuration patterns, *(ii)* expensive system calls, and *(iii)* imaging pipeline delay. Here, we discuss these sources of latency and propose mechanisms to mitigate their influence.

5.1 Sequential configuration patterns

The current Android camera system executes configuration through a step-by-step sequence, outlined in Section 4.1. The execution of one function blocks the next function, which contributes to reconfiguration latency. As an example, the current framework returns pending request frames to the application before deleting channels, streams, and buffers, using `waitUntilIdle` as a synchronization barrier, preserving previous resources while they are being used. As channel deletion and creation are grouped into the subsequent call to `configureStreams`, the HAL cannot create channels until after the HAL fully deletes previous channels. Thus, sequential configuration prevents new capture requests until completing previous pending requests.

5.1.1 Proposed alternative: Parallelize configuration procedures. Towards reducing the sequential bottleneck of reconfiguration, we propose to parallelize the deletion and creation of channels, streams, and buffers. While deletion must happen after pending requests return, *channel creation* does not need to wait for channel deletion or for the return of pending requests. Thus, we can parallelize the channel creation with the handling of previous requests, including the deletion of previous channels. This will allow channel creation and configuration to continue during `waitUntilIdle`.

5.2 Expensive system calls

The Android camera HAL invokes system calls to request kernel services, especially to delete, create, and initiate streams. Previous research shows that system calls are time consuming, due to context switches and I/O wait time [16][7]. Across the reconfiguration sequence, 79 ms is consumed by system calls, which contributes 29% to the frame-to-frame latency.

5.2.1 Proposed alternative: Reuse prior configurations to avoid system calls. We propose to give the HAL the controllability to hold and reuse prior channel configurations, especially for scenarios that repeatedly reuse configurations, e.g., toggling between low and high resolutions. This will avoid repeated use of system calls to delete and create channels. The proposed mechanism requires a rearchitecture of the HAL to maintain multiple configuration resources to swap them in as needed. We will study the memory implications of holding buffers and plan to devise low memory strategies, e.g., releasing unused buffers under memory pressure.

5.3 Imaging pipeline delay

The camera system uses a pipeline to capture and process image frames. This parallelizes several operations, including frame readout, white balancing, demosaicing, and RGB transformation. As these operations occur simultaneously, the pipeline exports frames at high frame rates.

However, there is substantial latency from capture request to frame return. For a 4-stage pipeline, this amounts to 132 ms of end-to-end latency. Moreover, as discussed in Section 5.1, sequential operation forbids requests from loading into the pipeline until the previous pipeline has been cleared. Thus, pipeline latency also contributes to frame-to-frame latency.

5.3.1 Proposed alternative: Timely HAL control of channel management. To keep the pipeline fully occupied, the previous session should continue capturing and processing frames until the camera hardware and processing stages have been configured for the new session. To provision for this, we propose to give the HAL precise timing control to issue system calls, rather than having the library dictate the scheduling. Enabled by our proposed parallel channel management (Section 5.1), the HAL can allow the previous session to continue while preparing new channel resources. After the HAL prepares the new channel, the HAL will trigger sensor drivers to capture at the new resolution, while simultaneously releasing the previous request channel. This will fully remove any influence of reconfiguration on frame-to-frame latency.

Potential impact of proposed mechanisms Our mechanisms will involve a redesign of the camera HAL and the camera framework library. However, as shown in Fig. 4, successful implementation will reduce end-to-end reconfiguration latency from 400 ms to 166 ms, allowing applications to flexibly adapt capture resolution to balance energy efficiency and task accuracy. More importantly, because our system preserves frame-to-frame latency, e.g., 33 ms for 30 FPS, applications will be able to reconfigure resolution without dropping frames. This ability will advance new classes of vision algorithms that use resolution-based tradeoffs to improve performance and efficiency in a variety of visual tasks.

6 RELATED WORK

Optimizing sensor hardware efficiency Sensor hardware efficiency limits vision applications. Jayasuriya et al. have presented a configurable hardware imaging signal processing pipeline [3]. RedEye proposes to push neural network processing to the sensor [11]. [9] has presented a novel framework to monitor and reduce sensors' contexts. Our work can interoperate with such optimizations, hiding the configuration delay of using these systems.

Operating systems optimization Prior work, such as RTDroid [18], has explored the deficiency of the Android system for providing real-time services. Others have proposed an energy optimization

framework enabling energy manager to utilize information provided by the application [10] . Inspired by them, we examine and characterize the capability of current systems for providing responsive and continuous service to sensor reconfiguration requests.

Exposing camera control Previous researchers have recognized the significance of programmability for camera operation, such as through Frankencamera [1] and OpenKCam [6]. Our proposed system integrates the idea of exposing low-level camera control to applications for providing a fine-grained camera control. We focus on reducing the latency for providing such configuration services.

7 CONCLUSION

To balance the energy efficiency of vision applications, the system should continuously reconfigure sensor operation. However, we observe that sensor resolution reconfiguration takes substantial latency during which no frame is returned to the application. We define and profile end-to-end and frame-to-frame latency caused by reconfiguring resolution. We identify three sources of latency: sequential configuration patterns, expensive system calls, and imaging pipeline delay. Built on these understandings, we propose a redesign of the camera system which will overcome the latency barrier caused by reconfiguring sensor resolution. We envision that our work will enable a future of adaptive vision on mobile devices.

Acknowledgement This material is based upon work supported by the National Science Foundation under Grant No. 1657602.

REFERENCES

[1] Andrew Adams, Eino-Ville Talvala, Sung Hee Park, David E. Jacobs, Boris Ajdin, Natasha Gelfand, Jennifer Dolson, Daniel Vaquero, Jongmin Baek, Marius Tico, Hendrik P. A. Lensch, Wojciech Matusik, Kari Pulli, Mark Horowitz, and Marc Levoy. 2010. The Frankencamera: An Experimental Platform for Computational Photography. In *Proceedings of the 37th International Conference and Exhibition on Computer Graphics and Interactive Techniques*.

[2] Brandon Amos, Bartosz Ludwiczuk, and Mahadev Satyanarayanan. 2016. *OpenFace: A general-purpose face recognition library with mobile applications*. Technical Report. CMU-CS-16-118, CMU School of Computer Science.

[3] Mark Buckler, Suren Jayasuriya, and Adrian Sampson. 2017. Reconfiguring the Imaging Pipeline for Computer Vision. *CoRR* (2017).

[4] Tiffany Yu-Han Chen, Lenin S. Ravindranath, Shuo Deng, Paramvir Victor Bahl, and Hari Balakrishnan. 2015. Glimpse: Continuous, Real-Time Object Recognition on Mobile Devices. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*.

[5] Sony Semiconductor Solutions Corporation. 2017. IMX377CQT ProductSummary v1.5. (2017).

[6] KHRONOS Group. 2013. Camera BOF. (2013). https://www.khronos.org/assets/uploads/developers/library/2013-siggraph-camera-bof/Camera-BOF_SIGGRAPH-2013.pdf

[7] Tomas Hruby, Teodor Crivat, Herbert Bos, and Andrew S. Tanenbaum. 2014. On Sockets and System Calls Minimizing Context Switches for the Socket API. In *Proceedings of the 2014 International Conference on Timely Results in Operating Systems*.

[8] Google Inc. 2017. Android Developers: CaptureResult. (2017). https://developer.android.com/reference/android/hardware/camera2/CaptureResult.html

[9] Seungwoo Kang, Jinwon Lee, Hyukjae Jang, Hyonik Lee, Youngki Lee, Souneil Park, Taiwoo Park, and Junehwa Song. 2008. SeeMon: Scalable and Energy-efficient Context Monitoring Framework for Sensor-rich Mobile Environments. In *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services*.

[10] L. Li, J. Wang, X. Wang, H. Ye, and Z. Hu. 2017. SceneMan: Bridging mobile apps with system energy manager via scenario notification. In *Proceedings of the 2017 IEEE/ACM International Symposium on Low Power Electronics and Design*.

[11] Robert LiKamWa, Yunhui Hou, Julian Gao, Mia Polansky, and Lin Zhong. 2016. RedEye: Analog ConvNet Image Sensor Architecture for Continuous Mobile Vision. In *Proceedings of the 43rd International Symposium on Computer Architecture*.

[12] Robert LiKamWa, Bodhi Priyantha, Matthai Philipose, Lin Zhong, and Paramvir Bahl. 2013. Energy Characterization and Optimization of Image Sensing Toward Continuous Mobile Vision. In *Proceedings of the 11th Annual International Conference on Mobile Systems, Applications, and Services*.

[13] ON Semiconductor 2017. *AR0330 1/3-inch CMOS Digital Image Sensor*. ON Semiconductor. Rev. 18.

[14] ON Semiconductor 2017. *MT9P031 1/2.5-Inch 5 Mp CMOS Digital Image Sensor*. ON Semiconductor. Rev. 10.

[15] Mahadev Satyanarayanan. 2015. A Brief History of Cloud Offload: A Personal Journey from Odyssey Through Cyber Foraging to Cloudlets. *GetMobile: Mobile Comp. and Comm.* 18 (2015).

[16] Livio Soares and Michael Stumm. 2010. FlexSC: Flexible System Call Scheduling with Exception-less System Calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*.

[17] Open Source Computer Vision. 2016. Real Time pose estimation of a textured object. (2016). http://docs.opencv.org/3.2.0/dc/d2c/tutorial_real_time_pose.html

[18] Yin Yan, Shaun Cosgrove, Varun Anand, Amit Kulkarni, Sree Harsha Konduri, Steven Y. Ko, and Lukasz Ziarek. 2014. Real-time Android with RTDroid. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*.