

Towards Fine-Grained Control of Visual Data in Mobile Systems

by

Jinhan Hu

A Dissertation Presented in Partial Fulfillment  
of the Requirements for the Degree  
Doctor of Philosophy

Approved February 2022 by the  
Graduate Supervisory Committee:

Robert LiKamWa, Chair  
Carole-Jean Wu  
Adam Doupé  
Suren Jayasuriya

ARIZONA STATE UNIVERSITY

May 2022

## ABSTRACT

With the rapid development of both hardware and software, mobile devices with their advantages in mobility, interactivity, and privacy have enabled various applications, including social networking, mixed reality, entertainment, authentication, and etc. In diverse forms such as smartphones, glasses, and watches, the number of mobile devices is expected to increase by 1 billion per year in the future. These devices not only generate and exchange small data such as GPS data, but also large data including videos and point clouds. Such massive visual data presents many challenges for processing on mobile devices. First, continuously capturing and processing high resolution visual data is energy-intensive, which can drain the battery of a mobile device very quickly. Second, data offloading for edge or cloud computing is helpful, but users are afraid that their privacy can be exposed to malicious developers. Third, interactivity and user experience is degraded if mobile devices cannot process large scale visual data in real-time such as off-device high precision point clouds.

To deal with these challenges, this work presents three solutions towards fine-grained control of visual data in mobile systems, revolving around two core ideas, enabling resolution-based tradeoffs and adopting split-process to protect visual data. In particular, this work introduces: (1) Banner media framework to remove resolution reconfiguration latency in the operating system for enabling seamless dynamic resolution-based tradeoffs; (2) LesnCap split-process application development framework to protect user’s visual privacy against malicious data collection in cloud-based Augmented Reality (AR) applications by isolating the visual processing in a distinct process; (3) A novel voxel grid schema to enable adaptive sampling at the edge device that can sample point clouds flexibly for interactive 3D vision use cases across mobile devices and mobile networks.

The evaluation in several mobile environments demonstrates that, by controlling

visual data at a fine granularity, energy efficiency can be improved by 49% switching between resolutions, visual privacy can be protected through split-process with negligible overhead, and point clouds can be delivered at a high throughput meeting various requirements. Thus, this work can enable more continuous mobile vision applications for the future of a new reality.

## ACKNOWLEDGMENTS

*To my beloved wife, who fills my boring PhD life with joy, peace, and love, who transforms me into a man, who sacrifices without complaints, and who trusts me and supports me with everything she has.*

*This dissertation and my success is all for you, my beloved wife, Xixi He.*

*Happy 3rd anniversary and the six years we have been together.*

*Looking forward to the adventure ahead of us.*

*To my parents, thanks for educating me and giving me a happy family and a competent platform.*

*Thanks for bearing with loneliness without me being around you. I hate separation and I wish we could meet again very soon.*

*To all my friends, thank you so much for the emotional support.*

*A special thanks to Hao Lu, from Chongqing to Arizona, from young to middle-age, our fourteen years friendship is more valuable than infinity gems. You are always a family member to me.*

*To my advisor, Robert LiKamWa, thank you so much for the opportunity. It's been such a wonderful journey with the Meteor Studio.*

*I wish you and the lab a prosperous future.*



## TABLE OF CONTENTS

	Page
CHAPTER	
1 INTRODUCTION .....	1
2 RELATED WORK .....	7
2.1 Continuous Mobile Vision .....	7
2.1.1 Dynamic Resolution-Based Tradeoffs .....	8
2.2 Protecting Data Privacy.....	9
2.2.1 Isolation and Compartmentalization .....	10
2.3 Streaming Dense Point Clouds .....	12
2.3.1 Collaborative System.....	14
3 SEAMLESS SENSOR RESOLUTION RECONFIGURATION FRAME- WORK .....	16
3.1 Introduction.....	16
3.1.1 Case Study for Resolution-Driven Tradeoffs .....	21
3.2 Background .....	22
3.2.1 Sequential Reconfiguration Process .....	24
3.2.2 Resolution Synchronization Creates Latency .....	25
3.2.3 Reconfiguration Latency Drops Frames .....	26
3.2.4 Design Guidelines .....	27
3.3 Design of Banner .....	28
3.3.1 Parallel Reconfiguration .....	30
3.3.2 Format-Oblivious Memory Management .....	34
3.4 Implementation of Banner .....	36
3.4.1 Parallel Reconfiguration .....	36
3.4.2 Format-Oblivious Memory Management .....	37

CHAPTER	Page
3.4.3	User Application Library ..... 38
3.5	Evaluation ..... 39
3.5.1	Evaluation Methodology ..... 40
3.5.2	Resolution Reconfiguration Latency Reduction ..... 42
3.5.3	Power Efficiency Improvement ..... 43
3.5.4	Implications ..... 46
4	SPLIT-PROCESS APPLICATION DEVELOPMENT FRAMEWORK .. 47
4.1	Introduction ..... 48
4.2	Background ..... 52
4.2.1	Mobile AR Development ..... 52
4.2.2	Permission Control ..... 53
4.2.3	Security Enforcement ..... 54
4.3	Threat and Trust Model ..... 54
4.4	Design of LensCap ..... 56
4.4.1	Enforced Split-Process Access Control ..... 57
4.4.2	Secured Communication Channels between Split Processes .. 58
4.4.3	Screen-Based Overlay Composition ..... 59
4.4.4	Fine-Grained Data Monitoring ..... 60
4.5	Programming Model ..... 61
4.6	Implementation of LensCap ..... 64
4.6.1	LensCap Permission Enforcement ..... 64
4.6.2	Split-Process for UE Development ..... 65
4.6.3	Secured Communication Channels ..... 66
4.6.4	Screen-Based Overlay Composition ..... 68

CHAPTER	Page
4.6.5	Fine-Grained Data Monitoring . . . . . 68
4.7	Evaluation . . . . . 70
4.7.1	Benchmark Applications . . . . . 71
4.7.2	Evaluation Metrics . . . . . 72
4.7.3	Application Performance . . . . . 74
4.7.4	Interactive Latency . . . . . 76
4.7.5	User Study . . . . . 77
5	EDGE-ASSISTED POINT CLOUD LIVE-CAPTURE AND STREAM- ING PLATFORM . . . . . 82
5.1	Introduction . . . . . 83
5.2	Motivation . . . . . 87
5.2.1	Point Cloud Data Distribution . . . . . 87
5.2.2	Voxel vs. Point vs. Octree . . . . . 89
5.2.3	Point Cloud Rendering . . . . . 90
5.2.4	Findings . . . . . 92
5.3	System Design . . . . . 93
5.3.1	Efficient Voxel Grid Schema . . . . . 93
5.3.2	Adaptive Sampling for Live-Streaming . . . . . 96
5.3.3	Point Cloud Resizing . . . . . 98
5.4	Implementation . . . . . 99
5.4.1	Multi-Threaded Data Pipeline . . . . . 99
5.4.2	Adaptive Point Cloud Sampling . . . . . 101
5.4.3	Point Cloud Resizing . . . . . 104
5.5	Evaluation . . . . . 105

CHAPTER	Page
5.5.1 The Performance of Adaptive Sampling .....	106
5.5.2 The Visual Quality of Point Cloud Resizing .....	111
6 CONCLUSION .....	112
REFERENCES .....	114

## Chapter 1

### INTRODUCTION

With the rapid development of both hardware and software, mobile devices such as smartphones are able to provide functionalities beyond simple texting and calling. For example, all smartphones nowadays are equipped with more than one high resolution image sensors, capable of capturing the world in tens of megapixels. By uploading those high resolution images to social media networks, people can share every detail in their life with their family and friends instantaneously, though possibly separated by a thousand miles physically. Other than static captures, the continuous capture of the physical world through those image sensors enable new realities such as Augmented Reality (AR). AR provides a unique interactive experience of virtual objects overlaying on top of the real-world environment. This unique interactive experience sparks new applications in many fields including education, entertainment, medicine, navigation, shopping, etc PAINE (2022). Beyond 2D captures, many smartphones are equipped with depth sensors that can capture the physical world in 3 dimensions (3D). For example, Apple recently deployed LiDAR sensors on iPhone, aiming at improving the understanding of the user's environment to improve its performance in tasks such as night vision, AR experiences, and machine learning Apple (2021); HUSSAIN (2021); Zhou and Tuzel (2017). The usage of 3D data such as point clouds portends continuing growth in more immersive applications, e.g., sports broadcasts Unity (2021). The varying user needs and application scenarios have reshaped mobile devices to many forms, including smart watches and smart glasses. As reported, the number of mobile devices will be increasing by 1 billion per year in the upcoming 5 years statista (2022).

The massive growth of mobile devices certainly eases people’s daily lives and brings people closer than ever. However, *mobile devices are able to generate massive visual data per second* which can create many challenging problems. First of all, mobile devices only have a limited battery life. Continuously capturing, processing, and rendering high quality visual data incurs high power consumption which will further drain the battery very quickly. In addition, the visual data may reveal private user information in a user’s visual environment to third party entities without the user’s knowledge. These private user information could contain personal belongings such as credit cards left on the table, business secrets on whiteboards, and a user’s facial identity. Finally, mobile devices cannot handle massive 3D visual data. Continuously capturing, transmitting, and rendering 3D visual data requires a high data bandwidth and a high processing capability. A delayed data processing will degrade the interactivity through mobile devices and the associated user experience.

Those challenges motivate the need of controlling visual data at an unprecedented fine granularity, for energy efficiency, visual privacy, and real-time interactivity. Towards fine-grained control of visual data, this work proposes three solutions, revolving around two key ideas, enabling runtime resolution-based tradeoffs and isolating the processing of visual data in a distinct process. These solutions include:

1. The Banner media framework, which is a seamless sensor resolution reconfiguration framework enabling mobile vision applications to dynamically adjust the capture resolution depending on various requirements to utilize the resolution-based tradeoffs at run-time to operate with an improved energy efficiency. This solution is described in §3.
2. The LensCap application development framework, which is built on top of split-process access control, allowing users with fine-grained and proactive control

over the app’s potential transmission of camera frames and the information derived from them. This solution is described in §4.

3. An edge-assisted multi-resolution adaptive point cloud live-capture and streaming framework tailored for interactive 3D use cases running on mobile devices. This framework revolves around an efficient voxel grid schema that represents point clouds in a set of voxel grids, each with a set of parameters that can be tailored to the needs of the users and their applications. This solution is described in §5.

For enabling runtime resolution-based tradeoffs in continuous mobile vision applications, this work advocates that mobile vision systems should be able to benefit from the ability to situationally sacrifice image resolution to save system energy when imaging detail is unnecessary. However, this work identifies that any change in sensor resolution leads to a substantial pause in frame delivery, caused by requiring user applications to frequently invoke a sequence of expensive system calls in order to request a new sensor resolution. Reconfiguring sensor resolution in the Android OS prevents the application from receiving frames for about 267 ms, the equivalent of dropping 9 frames (working at 30 FPS) from vision processing pipelines Hu *et al.* (2018). Consequently, computer vision applications don’t change resolutions at runtime, despite the significant energy savings at lower resolutions. Banner removes resolution reconfiguration latency through two techniques. Parallel reconfiguration maintains video capture streams and schedules sensor reconfiguration in parallel while the application is processing the frame. Format-oblivious memory management removes repeated memory allocation from the reconfiguration procedure, avoiding expensive system calls initiated by the application. Using these techniques, Banner completely eliminates frame-to-frame latency, allowing for seamless multi-resolution frame capture.

Banner also achieves the minimum possible end-to-end reconfiguration latency, fundamentally bounded by the pipeline latency of frame readout (usually larger than two frames). The reduction in reconfiguration latency results in a 49% power consumption reduction by reconfiguring the resolution from 1080p to 480p compared with computationally downsampling 1080p $\downarrow$ 480p, measured on a Jetson TX2 board.

For protecting visual privacy in cloud-based AR applications, this work identifies that under the current coarse-grained permission enforcement model, the internet permission, the camera permission, and the storage permission are all managed within one application process. Once granted, malicious developers of AR apps could silently collect camera frames and the information derived from them for malicious intent, including sending visual data to a private server, unbeknownst to the user. To address the privacy disclosure of continuous camera usage, this work introduces LensCap, an application development framework built on top of split-process access control, which allows users with fine-grained and proactive control over the app’s potential transmission of camera frames and the information derived from them. In LensCap, the split-process paradigm is adopted in the application layer, which is integrated into the app development flow. An AR application is split into a visual process with full access to operate on camera frames (but with network permission revoked) and a network process to maintain Internet communications (but with camera permission revoked), enforced by extending the legacy Android permission enforcement. LensCap enables both processes to present user interfaces through screen-based overlay composition. Then, data related to camera frames that need to be used in the network process can only be transmitted out of the visual process boundary through our trusted LensCap communication services, wrapped around trusted AR frameworks, and subject to the user’s monitoring and approval through LensCap data usage notifications at a fine granularity. This work prototypes LensCap as an Android library that can work



with standalone Android projects, as well as with Unreal Engine (UE) projects as a plugin. This work evaluates LensCap in five cloud-based AR applications that require the sharing of different types of image features, including camera pose, light estimation, point cloud, face region, and the camera frame. This work finds that the interactive latency between split processes and the overhead in app performance is negligible, even at 60 frames per second. Our user study further validates the performance similarity from the user’s perspective and the improvement in user confidence while using untrusted AR apps.

For enabling dense point cloud live-capture and streaming, this work identifies: (i) though raw point clouds are expensive to be streamed and processed, there is still a resolution-based tradeoff that can be utilized, especially when not all points are equally important. Capturing less points (subsampling) leads to less processing time and energy consumption but trades visual quality. (ii) Point representation is simply a special case of the voxel representation which presents opportunities to utilize the flexibility provided by the voxel representation to enable fast resolution-based tradeoffs. To this end, this work proposes an edge-assisted multi-resolution adaptive point cloud live-capture and streaming framework tailored for mobile platforms. The core idea behind our framework is to utilize the flexibility of the voxel representation to sample point clouds in different resolutions to exploit the resolution-based tradeoffs among visual quality, energy efficiency, and performance. Then, by fully utilizing edge computing, responsive high-quality visualization can be achieved. As a result, the system empowers users with the capability to visualize high quality point clouds captured off-device with the on-device level of mobility. This enables a variety of interactive apps including AR-based remote coaching, immersive sports viewing, and 3D model reconstruction, etc. Our framework is prototyped using three off-the-shelf devices, consisting of an Azure Kinect to capture high resolution/precision point clouds,

a Jetson TX2 development board to simulate the early data analysis and preparation for utilizing real-time resolution-based tradeoffs on the mobile camera device, and a smartphone to run an interactive AR application. Despite being evaluated with specific hardware, our framework can work on other platforms, including other depth sensors, edge computing platforms, and mobile devices. Our framework is evaluated from two aspects. First, this work evaluates the effectiveness of our adaptive sampling component in terms of the data throughput, the “compression” ratio, and the energy efficiency. Second, this work evaluates the visual quality of our point cloud resizing component in terms of SSIM. The evaluation demonstrates that our edge-assisted framework is able to deliver point clouds at a comparable high throughput meeting various real-time requirements with an improved performance, compression ratio, and energy efficiency, while maintaining a reasonable visual quality.

The rest of this paper is organized as follows: §2 introduces related work, §3 presents Banner media framework, §4 presents LensCap application development framework, §5 describes our point cloud live-capture and streaming platform, and §6 concludes this work.

## Chapter 2

### RELATED WORK

Fine-grained control of visual data is always desired, but realizing it presents many challenges. In this chapter, related work is discussed in detail from three aspects, maintaining continuous mobile vision, protecting data privacy, and streaming dense point clouds to mobile clients.

#### 2.1 Continuous Mobile Vision

The limited energy (battery life) is always a problem for mobile devices. If vision applications are kept working at a low energy efficiency, the uniqueness of mobility and convenience brought by mobile devices will be impaired. Towards continuous mobile vision applications, there are many different types of solutions. Hegarty *et al.* (2016) introduced a flexible multi-rate image processing pipeline to improve vision tasks' performance with three orders of magnitudes lower energy consumption. Hegarty *et al.* (2014) introduced Darkroom to ease developers burden when they want to utilize specialized image signal processors for higher energy efficiency. Roy *et al.* (2011) introduced an energy-efficiency context recognition framework for multi-modal sensing. Priyantha *et al.* (2011) proposed a dedicated low-level processing hardware to reduce sensing power. Ben Abdesslem *et al.* (2009) presented Abdesslem as the solution in a multi-sensory environment in which the sensing cost is reduced by using cheaper sensors more often. Lin *et al.* (2012) presented Reflex to help developers leverage low-power processors on mobile devices. Buckler *et al.* (2017) introduced a reconfigurable software pipeline to replace image signal processing hardware to reduce energy consumption. Chen *et al.* (2016) proposed Glimpse, which is an energy

efficient continuous object detection system that is able to balance the accuracy and energy tradeoff between local and offload computing. LiKamWa and Zhong (2015) introduced Starfish to enable resource sharing among computer vision applications to improve energy efficiency. LiKamWa *et al.* (2016) presented RedEye which aims at shifting the early vision processing to analog domain to reduce image sensing power consumption. Similar to those works, this work also aims at improving the runtime energy efficiency of continuous mobile vision applications by tackling the root cause of a long latency during resolution change in the operating system.

### 2.1.1 Dynamic Resolution-Based Tradeoffs

Resolution-based tradeoffs in task accuracy, processing latency, and energy efficiency can be identified not only in 2D visual data, but also 3D point clouds. These tradeoffs have been utilized in a variety of continuous mobile vision applications for energy efficiency Likamwa *et al.* (2021); Priyantha *et al.* (2011); Chu *et al.* (2011); Haris *et al.* (2018a); LiKamWa *et al.* (2013a). In 2D visual data, image downsampling has been a long term choice (either by skipping or averaging surrounding pixels) to reduce the data size for various purposes including more efficient computation. System-wise, Kodukula *et al.* (2021) introduced Rhythmic Pixel Regions to encode pixels coming out of the image sensor into regions represented by different resolutions before they even reach the memory such that the memory footprint is reduced while with the task accuracy maintained. In addition, Hu *et al.* (2018) demonstrated that reconfiguring the sensor resolution physically is more energy efficient than downsampling. However, unlike computational photography tasks such as HDR imaging which can blend several frames captured with different sensor settings for improving image quality, no frame-critical application is able to change resolution at runtime because resolution change incurs long latency penalties in current operating systems. In 3D

visual data, Riegler *et al.* (2016) proposed OctNet, which utilizes an unbalanced Oct-tree structure to represent a region of points with different resolutions based on its importance such that the overall orientation estimation accuracy in the point cloud-based deep learning application can be improved. Wang *et al.* (2020) explored the representation of LiDAR data in the context of object detection in which they allow fixed-size voxels to be reconfigured with different resolutions determined by their neighbors. In this work, we also advocate that the resolution-based tradeoff is the key enabler not only to improving energy efficiency in traditional AR applications, but also to interactive use cases in 3D vision applications on mobile devices.

## 2.2 Protecting Data Privacy

Protecting data privacy on mobile devices is always a research problem. Data privacy is usually protected by controlling the input, monitoring the usage, and tracking the output. For example, Arzt *et al.* (2014) presented FlowDroid, a static taint-analysis system to address data leakage in malicious applications. FlowDroid can model the complete lifecycle of an Android application and precisely monitor contexts, flows, fields, and objects with affordable performance overhead through on-demand alias analysis. Enck *et al.* (2010) introduced TaintDroid to taint, analyze, and track user’s sensitive information at the granularity of variables, messages between applications, native methods, and files. TaintDroid is able to dynamically track those tainted data and identify how they are impacting other data that might cause data leakage. Wang *et al.* (2019) proposed LeakDoctor, a system that automatically detects privacy disclosure and determines if those privacy disclosures are necessary for functionalities of the app. However, these types of solutions discussed above can hardly be applied to protect visual privacy because of the high cost (in terms of latency) of analyzing visual data at runtime, which usually incurs a large throughput.

## Protecting Visual Privacy

Many previous works attempt to protect visual privacy by controlling the input, i.e., by depriving untrusted vision applications from accessing the whole camera frame Roesner *et al.* (2014); Aditya *et al.* (2016); Olejnik *et al.* (2017). Jana *et al.* (2013) introduced the Darkly system to address the threat of data over-collection and aggregation in untrusted third-party vision applications. In Darkly, camera frames are turned into opaque references and untrusted vision applications can only dereference them through trusted library APIs. Similarly, the Oculus Quest camera system Facebook (2021) directly prohibits apps from accessing the passthrough camera. Instead, developers are only able to utilize camera poses, controller poses, and hand poses. In Raval *et al.* (2014) and Raval *et al.* (2016), Raval *et al.* provided tools to give AR users finer-granularity control over their camera frames. AR users are able to define the part of camera frames that can be seen by untrusted vision applications. These types of works are inflexible and insufficient to protect visual privacy in cloud-based AR apps. Depriving vision applications from accessing the whole camera frame might limit some vision apps that do require to work on objects and features of a camera frame directly and render the results or restrict AR experiences because AR applications need to provide the whole camera frame for virtual object overlay. Furthermore, visual frames are still stored and managed within one application process, which remain vulnerable to be collected through network access.

### 2.2.1 Isolation and Compartmentalization

Isolation and compartmentalization are adopted in both hardware and software domains to separate the execution of untrusted code from trusted code Shekhar *et al.* (2012); Zhang *et al.* (2013); Huang *et al.* (2017); Backes *et al.* (2015); Jensen *et al.*

(2019); Hu *et al.* (2021a). Herbster *et al.* (2016) proposed Privacy Capsules which targets on protecting the leakage of user information through untrusted third-party applications. Privacy Capsules enforces applications to first execute in the unsealed phase in which the application has no access to the sensitive input but full access to the network resource, and then in the sealed phase in which the application gains access to the sensitive input but losing the capability of network communications. Raval *et al.* (2019) proposed to isolate plugins from the application as an individual app. It allows users to mediate resource requests made by apps which further enables more flexible authorizations to them. Dawoud and Bugiel (2019) in DroidCap also pointed out the necessity of application compartmentalization to protect privacy. DroidCap, which is a system that associates each IPC object with permissions for capability-based access control, could be integrated with our work for capability-based access control between split processes. Track and Kilpatrick (2003) introduced Privman as a C library for partitioning applications in UNIX environment to ease developer’s burden when developing partitioned applications. In Privman, developers need to separate their applications into a privilege server process and a main application process, in which the main application process only has limited privileges. Apart from software solutions, Intel introduced Software Guard Extensions (SGX) Costan and Devadas (2016) to allow users to define private regions of memory for secured execution, which further inspired tons of security and privacy works Sanchez Vicarte *et al.* (2020); Shen *et al.* (2020). The idea of isolation and compartmentalization presented in those systems can largely be borrowed to protect visual privacy in cloud-based AR applications. In particular, a solution needs to be developed to isolate potentially malicious behavior from accessing visual data streams, without affecting the sensitive user experience.

## 2.3 Streaming Dense Point Clouds

Many previous works tried to stream dense point clouds at real-time. In general, they can be categorized into two types: compression-based and study-based. Both types of optimizations capture point clouds first and then perform operations on them separately in an offline stage.

### **Compression-Based Optimization**

In this category, many works represent the sparse 3D point clouds in tree structures to enable real-time optimizations and efficient operations. Among them, Draco Google (2021b) and PCL Rusu and Cousins (2011) are the two most popular libraries for 3D data processing. In Draco, 3D points are compressed using the KD-tree structure (lossless). Meanwhile, in PCL, points are compressed with the Octree structure (lossy). Both libraries can compress point clouds to be magnitudes lower for optimization in storage and transmission. With the help from those libraries, Lee *et al.* (2020) further introduced GROOT system to advance the conventional sequential Octree into a Parallel-Decodable tree. GROOT uses Octree breadth bytes and depth bytes to independently encode each node in the last three depth layers, through which the decoding latency can be drastically reduced. In Kammerl *et al.* (2012), on top of spatial redundancy removed by utilizing Octree, Kammerl *et al.* proposed a double-buffering encoding and decoding scheme to track the correspondence of points in different point clouds such that temporal redundancy can also be removed for realizing real-time point cloud compression on a PC.



## Study-Based Optimization

In this category, researchers perform studies on either the point cloud data itself or the user’s behavior during point cloud visualization. Then, point clouds are captured and stored in a local or remote server and then optimized and streamed to mobile users guided by those study results, e.g., Qian *et al.* (2019). Qian *et al.* (2018) proposed Flare to segment, fetch, and deliver 360°-video in tiles, based on the prediction of user’s viewport trained on a large dataset. Similarly, Han *et al.* (2020) presented ViVo to deliver high quality point cloud data to mobile client devices. ViVo predicts user’s viewport and optimizes point cloud data based on viewport visibility, occlusion visibility, and distance visibility such that the bandwidth for streaming point cloud data is reduced with its visual quality preserved. He *et al.* (2018) introduced Rubiks framework, which splits the volumetric video into high-low quality tiles based on the probability of being viewed by users, such that even 8K video streams can be streamed at real-time. Feng *et al.* (2020) proposed a framework to remove both the temporal and spatial redundancy in streaming LiDAR point clouds, achieving a higher compression rate and a lower computing complexity.

These two types of optimizations are not sufficient for three reasons. First, the high latency incurred to compress and study point clouds is not viable for its live-capture, streaming, and rendering on mobile client devices. The detachment of capture and streaming degrades the user experience in a variety of interactive use cases. Second, viewport prediction yields relatively low accuracy dealing with real-time data. The user experience will again be degraded if wrong portion of the point cloud data is captured, streamed, and rendered. Third, computing and storing large scale point clouds is very challenging, especially when we continue to push the computing to the edge or to the camera device itself. A layered storage solution needs to be developed

to distribute computing to different components to increase the overall throughput for an improved user experience.

### 2.3.1 Collaborative System

Other than these solutions discussed above, we can rely on edge or collaborative systems to process heavy point clouds for real-time performance AlDuaij *et al.* (2019); Zhu *et al.* (2020), as edge computing is becoming more popular and available. Qiu *et al.* (2018) introduced AVR system for autonomous driving cars to share and work on point clouds collaboratively at runtime such that the visibility of cars can be drastically improved. Zhang *et al.* (2021b) introduced Elf, which is a system that offloads smaller inference tasks in parallel across multiple edge servers such that the system can operate on high resolution images. In recent years, the emergence of IoT devices, as well as technologies such as AR/VR incurs huge data throughput which further pushes the data and the computing boundary to the edge Liu and Gruteser (2021); Jiang *et al.* (2021); Zhang *et al.* (2019, 2021a,c). Liu *et al.* (2019) presented a framework which encodes higher interests RoIs with higher quality and streams and infers on frame slices in parallel to reduce latency in offloading and improve detection accuracy in edge assisted object detection use cases. Ben Ali *et al.* (2020) proposed an edge-assisted SLAM system which decouples the mapping operation by utilizing a layered map storage (global map on the edge and local map on the device) to improve performance. He *et al.* (2021) introduced VI-Eye, an infrastructure-assisted framework which efficiently extracts a small sets of saliency points from key semantic objects to achieve real-time registration of two point clouds in autonomous driving.

We also advocate that edge computing will be the key enabler to future dense point cloud-based interactive use cases on mobile devices since the uniqueness of edge computing in providing faster responses, as well as layered storage and computing

resources. However, solutions need to be developed to find a balance among streaming latency, computing latency, throughput expectation, and interaction requirements, with different types of data combined and distributed to be processed on different network components.

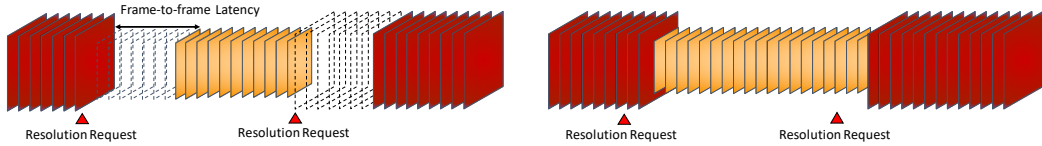
### SEAMLESS SENSOR RESOLUTION RECONFIGURATION FRAMEWORK

Mobile vision systems would benefit from the ability to situationally sacrifice image resolution to save system energy when imaging detail is unnecessary. Unfortunately, any change in sensor resolution leads to a substantial pause in frame delivery – as much as 280 ms. Frame delivery is bottlenecked by a sequence of reconfiguration procedures and memory management in current operating systems before it resumes at the new resolution. This latency from reconfiguration impedes the adoption of otherwise beneficial resolution-energy tradeoff mechanisms.

We propose Banner as a media framework that provides a rapid sensor resolution reconfiguration service as a modification to common media frameworks, e.g., V4L2. Banner completely eliminates the frame-to-frame reconfiguration latency (226 ms to 33 ms), i.e., removing the frame drop during sensor resolution reconfiguration. Banner also halves the end-to-end resolution reconfiguration latency (226 ms to 105 ms). This enables a more than 49% reduction of system power consumption by allowing continuous vision applications to reconfigure the sensor resolution to 480p compared with downsampling 1080p $\downarrow$ 480p, as measured in a cloud-based offloading workload running on a Jetson TX2 board. As a result, Banner unlocks unprecedented capabilities for mobile vision applications to dynamically reconfigure sensor resolutions to balance the energy efficiency and task accuracy tradeoff.

#### 3.1 Introduction

The high energy consumption of visual sensing continues to impede the future of mobile vision in which devices will continuously compute visual information from



(a) In legacy systems, any change in sensor resolution leads to a substantial pause in frame delivery.

(b) Banner completely removes frame-to-frame latency for reconfiguring sensor resolution.

Figure 3.1: Compared with current systems, Banner enables rapid and seamless sensor resolution reconfiguration.

sensory data, e.g., for visual personal assistants or for augmented reality (AR). While vision algorithms continue to improve in task accuracy and speed, mobile and wearable vision systems fail to achieve sufficient battery life when vision tasks are continuously running. Continuous video capture drains the battery of Google Glass in 30 minutes Berenbaum (2013).

It is well known that a common culprit is the energy-expensive traffic of image data Kodukula *et al.* (2018); LiKamWa *et al.* (2013b). Transferring high resolutions at high frame rates draws substantial power consumption from the analog-digital conversion, the sensor interface transactions, and the memory usage. Simply capturing 1080p frames at 30 frames per second consumes more than 2.4W of system power measured on a Moto Z smartphone. However, capturing and displaying 480p frames only consumes 1.3W of system power.

Image resolution can create an interesting tradeoff for visual tasks: low resolution promotes low energy consumption, while high resolution promotes high imaging fidelity for high visual task accuracy. For example, as we explore with our AR marker-based pose estimation case study (§3.1.1), lower resolutions suffice when an AR marker is close, but high resolutions are needed when the AR marker is far away

or small. This tradeoff has been explored by several visual computing system works including marker pose estimation, object detection, and face recognition Ha *et al.* (2014); Hu *et al.* (2018); LiKamWa *et al.* (2013b); Haris *et al.* (2018b); Redmon and Farhadi (2018); Lin *et al.* (2017); Zhang *et al.* (2018); Ruzicka and Franchetti (2018); Buckler *et al.* (2018). We too advocate that mobile vision systems should be able to benefit from the ability to situationally sacrifice image resolution to save system energy when imaging detail is unnecessary.

**Unfortunately, any change in sensor resolution leads to a substantial pause in frame delivery.** This is illustrated in Figure 3.1a. We measure that reconfiguring sensor resolution in the Android OS prevents the application from receiving frames for about 267 ms, the equivalent of dropping 9 frames (working at 30 FPS) from vision processing pipelines Hu *et al.* (2018). Consequently, computer vision applications don’t change resolutions at runtime, despite the significant energy savings at lower resolutions. For example, Augmented Reality applications such as “Augment” and “UnifiedAR” constantly work at 1080p, drawing 2.7 W of system power.

Thus, in this paper, we target image sensor *resolution reconfiguration latency* as a chief impediment of energy-efficient visual systems. Referring to Hu *et al.* (2018), we break the resolution reconfiguration latency into two types of latency. *End-to-end reconfiguration latency* is the time between an application’s request to change resolution and the time the application receives a frame of the new resolution. *Frame-to-frame latency* is the interval between two frames provided to the application in which the latter frame is configured at the new resolution.

The problem of long resolution reconfiguration latency is common across all mobile platforms, as we measured on different devices. In the Android OS, there is a 400 ms end-to-end reconfiguration latency Hu *et al.* (2018). In the Linux V4L2 media

framework, we observe a 260 ms end-to-end reconfiguration latency. End-to-end reconfiguration latency in iOS also takes around 400 ms, as measured by timestamping a simple video capturing application we built in Xcode Ibrahim (2019). Similarly, end-to-end reconfiguration latency in Gstreamer with Nvidia Libargus occupies more than 300 ms.

Almost all of the resolution reconfiguration latency originates from the operating system; at the sensor level, hardware register values are effective by the next frame ON Semiconductor (2017a,b); Apple (2018). We proposed several alternatives at the Android framework and HAL level for discussion in our previous work Hu *et al.* (2018). However, from deeper understanding of the Android OS, we find that the problem stems from the lower level system, i.e., media frameworks in the kernel. The underlying issue is that the kernel’s media frameworks require user space applications to frequently invoke a sequence of expensive system calls in order to request a new sensor resolution.

Our study of the media frameworks exposes several key insights. First, the current streaming pipeline needs to be preserved during resolution reconfiguration. Frames already captured at the previous resolution are useful and need to be read out. Second, resolution change should also be immediately effective in the next capture. This capture will be available after moving through the pipeline. Third, synchronizing the resolution of frame buffers across the system stack is expensive and should be avoided. As it stands, media frameworks require the application to initiate expensive system calls to repeatedly allocate memory for the frame buffers.

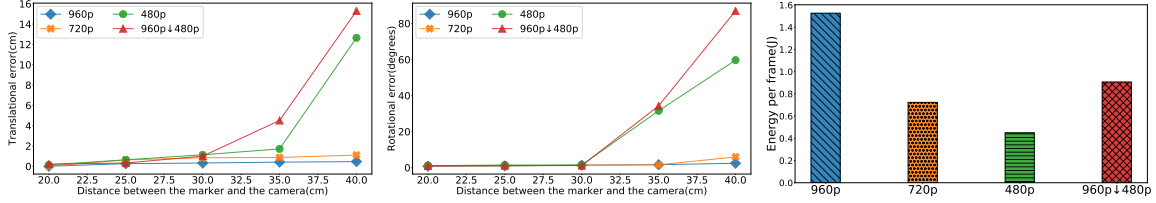
To exploit these key insights, we design Banner: a system solution for rapid sensor resolution reconfiguration. Banner revolves around two techniques. *Parallel reconfiguration* maintains video capture streams and schedules sensor reconfiguration in parallel while the application is processing the frame. *Format-oblivious memory*

*management* removes repeated memory allocation from the reconfiguration procedure, avoiding expensive system calls initiated by the application. Using these techniques, Banner completely eliminates frame-to-frame latency, as illustrated in Figure 3.1b, allowing for seamless multi-resolution frame capture. Banner also achieves the minimum possible end-to-end reconfiguration latency, fundamentally bounded by the pipeline latency of frame readout (usually larger than two frames). In extreme cases, if the application requests only one buffer to be allocated (not allowed for video streaming) or if the system allows frames already captured to drop, the end-to-end reconfiguration latency can further be reduced in Banner.

Because of the unavailability of open-source camera drivers and camera host drivers for Android devices, our Banner prototype is implemented in the Linux kernel. We evaluate Banner’s efficacy within the Linux V4L2 framework by running three workloads on a Jetson TX2 board with the Ubuntu system, including display-only, cloud-based offloading, and marker-based pose estimation. Our evaluation confirms that Banner completely eliminates frame-to-frame latency, even for workloads operating at 30 FPS. Furthermore, Banner creates a 54% reduction in end-to-end reconfiguration latency (from 226 ms to 105 ms).

The reduction in reconfiguration latency results in a 49% power consumption reduction by reconfiguring the resolution from 1080p to 480p compared with computationally downsampling 1080p $\downarrow$ 480p, measured on a Jetson TX2 board. While we implement and evaluate our design choices based on Linux V4L2, they can be generalized to other media frameworks, such as Gstreamer (which can capture videos from V4L2 devices), and Linux-based operating systems, including the Android OS. Altogether, Banner will unlock new classes of vision algorithms that can balance the resolution-based energy efficiency and accuracy tradeoffs to maximize performance in a variety of continuous mobile vision tasks.





(a) Low resolution maintains a low translation error when marker is close to camera. (b) Low resolution maintains a low rotation error when marker is close to camera. (c) System energy consumption at 480p is 70% and 50% less than 960p and downsampling.

Figure 3.2: In our marker-based pose estimation case study, task accuracy (translation and rotation error) can be maintained and system energy consumption (energy per frame) can be reduced by 70% if sensor resolution is reconfigured from 960p to 480p when the distance between the marker and the camera is reduced from 35 cm to 20 cm.

### 3.1.1 Case Study for Resolution-Driven Tradeoffs

To motivate our work, we present a case study around a marker-based pose estimation application running on a Moto Z mobile phone. Marker-based pose estimation forms the foundation for many AR frameworks, including Vuforia Vuforia (2021), ARCore Google (2021a), and ARKit Apple (2021) for image-based tracking. Our exploration of marker-based pose estimation allows us to analyze the resolution-based energy and accuracy tradeoff in mobile vision tasks. The pose estimation application uses an ORB feature detector, Flann-based matcher, and Perspective-n-Point algorithm to detect keypoints in an image frame, match keypoints with model descriptors, and estimate the position of the virtual camera against the physical environment respectively on a frame-to-frame basis.

The energy efficiency is characterized by the power traces acquired from the Trepn Profiler and the number of frames processed per second measured at different reso-

lutions. To evaluate task accuracy, we use the MSE rotation and translation vector errors compared with the “ground truth” acquired from the highest resolution. Prior work Hu *et al.* (2018) has already demonstrated that based on the distance and viewing angle between the camera and the marker, sensor resolution needs to be actively reconfigured to balance efficiency and performance. Similarly, results in Figure 3.2 show that, by reconfiguring the sensor from 960p to 480p while the sensor is approaching the marker from 35 cm to 20 cm, the task accuracy can be maintained (Figure 3.2a and 3.2b) and a 70% energy consumption reduction can be achieved (Figure 3.2c).

As an alternative to changing sensor resolution, the system can computationally downsample the frames to reduce the computational workload of the vision algorithm. However, results in Figure 3.2c show that, capturing at 480p costs almost 50% less energy than computational downsampling 960p $\downarrow$ 480p, not to mention the higher task accuracy.

In conclusion, physically reconfiguring the sensor resolution is the most viable way to balance the resolution-based energy efficiency and task accuracy tradeoff for continuous mobile vision tasks. However, sensor resolution reconfiguration is limited by a substantial latency.

## 3.2 Background

In this section, we elaborate on how user applications request different sensor resolutions using the Video4Linux2 (V4L2) framework. The V4L2 framework provides APIs for applications to manipulate cameras on Linux. V4L2 is commonly used by almost all Ubuntu desktops and Android devices built upon the Linux system.

In V4L2, image sensor resolution reconfiguration follows a strict sequential procedure. This sequential procedure leads to a substantial amount of end-to-end and frame-to-frame reconfiguration latency, which impedes the ability for applications

to utilize resolution-based energy tradeoffs. Through this section, we explore the V4L2 implementation on an NVIDIA Jetson TX2 board with an ON Semiconductor AR0330 sensor.

## V4L2 System Architecture

In the V4L2 framework, there are four main driver modules in the kernel that collaborate to provide camera services. The **V4L2 driver** is responsible for exposing camera control operations to the user application, such as opening the V4L2 camera or setting its exposure or brightness. The **camera host driver**, which implements the V4L2 driver and V4L2 camera interfaces, is responsible for ensuring the proper input and output format for frames flowing between the camera and the memory, as well as starting and stopping the reception of camera frame data. The **video buffer driver**, which is a helper to the V4L2 driver, is responsible for allocating and deallocating buffers with proper sizes for corresponding resolution requests. The **camera driver**, which communicates with the camera hardware, is responsible for setting up the camera for the requested output.

Before the application can start streaming with the V4L2 camera devices, it needs to request at least a pair of buffers to store and process the captured frames. The buffer ownership transfer is realized by dequeuing buffers `ioctl(VIDIOC_DQBUF)` and queuing buffers `ioctl(VIDIOC_QBUF)` between the application and the camera host driver. The application dequeues a buffer when the capture is completed by the image sensor. The application queues a buffer back for sensor capture after the processing on it is done – the application relinquishes control of the buffer. Depending on the needs of the imaging pipeline, the application can require more buffers, such that multiple pipeline stages can simultaneously address buffers. All buffers ready for applications to read are stored in the camera host driver. Typically, only one buffer is transferred

to the application at a time.

### 3.2.1 Sequential Reconfiguration Process

Once the video stream has been started, V4L2 requires the application to reconfigure sensor resolution in a sequence of steps. Notably, each subsequent step in the sequence invokes a different subsystem, creating synchronization issues. We illustrate this sequential procedure in Figure 3.4a and detail it here.

1. The application initializes a resolution request while the camera is capturing.
2. The application calls V4L2 `ioctl(VIDIOC_STREAMOFF)`, which is implemented in the camera host driver and the camera driver to turn off current working streams. This step takes around 75 ms.
3. The application calls `munmap()`, which is implemented in the video buffer driver to deallocate the memory. This step takes less than 1 ms.
4. The application calls V4L2 `ioctl(VIDIOC_S_FMT)`, which is implemented in the camera host driver and the camera driver to set the sensor's output format.
5. The application calls V4L2 `ioctl(VIDIOC_REQBUFS)` and `mmap()`, which are implemented in the video buffer driver to request, allocate, and map new sets of buffers. Together with step 4, initializing the device takes around 31 ms.
6. The application finally calls V4L2 `ioctl(VIDIOC_STREAMON)`, which is implemented in the camera host driver and the camera driver to set the input and output format of the channel and then start the video stream. This step takes around 72 ms.
7. The first frame at new resolution is returned after a pipeline latency, typically several frame times later, depending on the pipeline depth.

Reconfiguration operation	Average execution time
Stop streaming	75 ms
Initialize device	31 ms
Start streaming	72 ms

Table 3.1: Expensive operations and their average cost in current sensor resolution reconfiguration procedure measured on the TX2/AR0330 setup.

Table 3.1 shows the latency costs of several expensive operations in the sensor resolution reconfiguration procedure measured on the TX2/AR0330 setup.

### 3.2.2 Resolution Synchronization Creates Latency

Throughout the reconfiguration process, there are several strict resolution synchronizations among the camera host driver, the video buffer driver, and the camera driver, each of which introduces a substantial reconfiguration latency.

First, resolution synchronization between the video buffer driver and the camera driver is established by requesting buffer size based on specific sensor format. This synchronization ensures that there will be enough frame buffer space to hold complete frames. If the syscall `ioctl(VIDIOC_S_FMT)` is called to set the sensor resolution, `ioctl(VIDIOC_REQBUFS)` and `mmap()` also need to be called for a new set of buffers.

Second, resolution synchronization between camera host driver and camera driver is established by updating the camera driver host state based on the camera’s format. If `ioctl(VIDIOC_S_FMT)` is called to set the sensor resolution, the input state of camera host driver also needs to be reconfigured. This synchronization ensures that the video input module on board has the proper format to receive frames flowing from the camera.

Third, the previous two synchronizations force a resolution synchronization between the camera host driver and video buffer driver. That is, if the system requires a

new set of buffers, the output state of camera host driver also needs to be reconfigured.

By synchronizing resolution among these drivers, the camera service ensures correct capture, delivery, and render of image frames. But this strong resolution coupling among drivers creates bottlenecks; if an application requests a new resolution, the whole configuration procedure described above in §3.2.1 will be invoked, creating a substantial latency.

### 3.2.3 *Reconfiguration Latency Drops Frames*

As shown in Figure 3.7, the overall end-to-end reconfiguration and the frame-to-frame latency are both about 230 ms in the legacy V4L2 framework, as we measured on the TX2/AR0330 system. For a camera running at 30 FPS, a 230 ms frame-to-frame reconfiguration latency is equivalent to the system dropping 8 camera frames. In addition, the legacy V4L2 framework abandons all captured frames that are stored in buffers once the application requests a new sensor resolution. Thus, depending on how many buffers are requested by the application ( $N$ ), the number of frames dropped could be  $N + 8$ . The number of requested buffers ( $N$ ) must be larger than 2 (typically 3 or 4) The kernel development community (2019).

We see resolution reconfiguration latency manifested on all devices we tested. We measure that end-to-end resolution reconfiguration latency in Android and iOS devices both consume about 400 ms. A fast sensor resolution reconfiguration solution needs to be introduced to media frameworks so that frame-critical computer vision applications on top of them can frequently reconfigure the sensor resolution to improve energy efficiency.

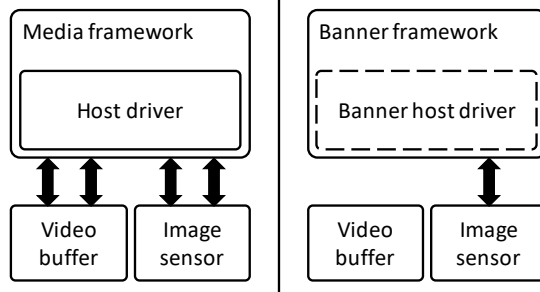


Figure 3.3: Banner helps the application reduce the number of required system calls to reconfigure sensor resolution to one `ioctl(VIDIOC_RECONFIGURE)` call, instead of multiple systems calls: `mmap()`, `munmap()`, `ioctl(VIDIOC_STREAMON)`, and `ioctl(VIDIOC_STREAMOFF)`.

### 3.2.4 Design Guidelines

We declare the following insights for inspiring the design of a rapid and seamless sensor resolution reconfiguration system:

(i) Preserve the pipeline of existing frames. Frames already captured and stored in the pipeline are still meaningful. The legacy V4L2 framework abandons those frames to fulfill the new resolution request immediately. On the contrary, the Android OS will issue the new resolution request only after pipelined frames are processed and delivered properly. For some visual tasks – including marker-based pose estimation – every frame is critical to task performance because of the potential negative influence on task accuracy and user experience. The sensory data should be continuous, i.e., frame drop is unacceptable. The system should find a way to maintain current streams while reconfiguring the sensor for new resolution.

(ii) Resolution change should be immediately effective in the next capture. Sensor register changes can be effective in the next capture, as is done for setting up different exposure time for consecutive capture requests in HDR mode Apple (2018).

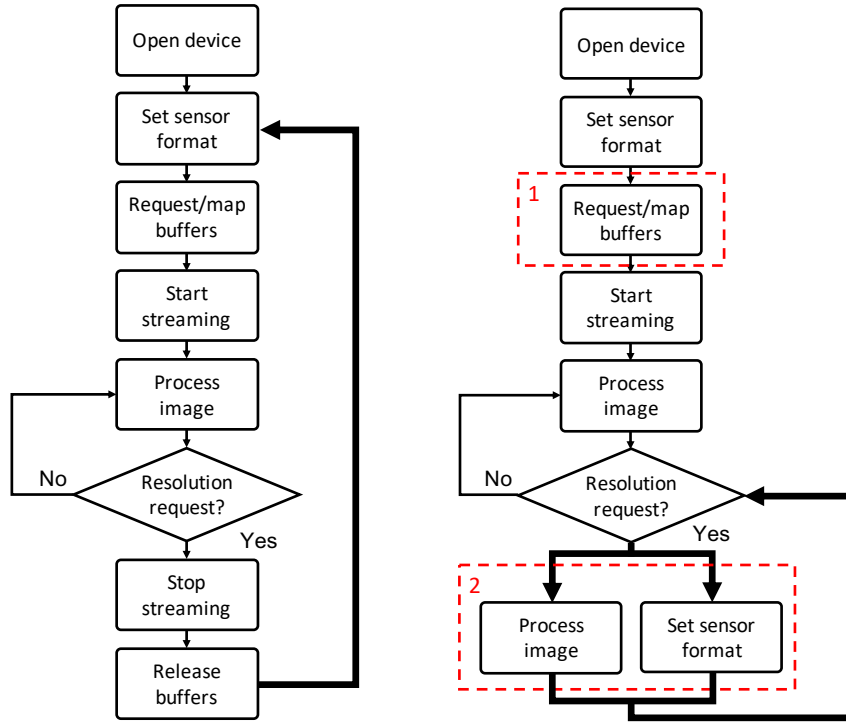
Similarly, the system should reconfigure related sensor registers immediately and asynchronously once there is a new resolution request. This would allow applications to expect and utilize the prompt resolution change.

(iii) Minimize synchronization across the video system stack, while ensuring correct sensory data. Resolution synchronizations among different driver modules lead to repeated sequential reconfiguration every time there is a new resolution request which causes huge amount of latency. In addition, resolution synchronizations trigger some expensive and redundant system calls initiated by the application, including `mmap()`. As long as the application knows the resolution of the frames it is processing and the sensor knows the resolution for each frame it is capturing, the data will be correctly delivered and interpreted. We argue that buffer size synchronization between the sensor and the application is unnecessary. Memory management can be oblivious to format.

### 3.3 Design of Banner

Built on these derived design guidelines, we design Banner to address the resolution reconfiguration latency problem in the legacy V4L2 framework. Banner is a fast sensor resolution reconfiguration framework that can provide frames continuously, even between two frames at different resolutions. While we design Banner to inter-operate with the V4L2 framework, the underlying concepts are generic to all media frameworks. Compared to resolution reconfiguration in today’s system, Banner halves the end-to-end reconfiguration latency and completely removes the frame-to-frame reconfiguration latency, i.e., no frame drops. As a result, Banner unlocks a variety of continuous mobile vision applications to control their image sensors for desired resolutions. This allows new potential to balance energy efficiency and accuracy tradeoffs.





(a) Resolution reconfiguration in legacy V4L2      (b) Resolution reconfiguration in Banner

Figure 3.4: In Banner, most of the sequential procedures are avoided for reconfiguring sensor resolution. (i) Banner avoids repeated memory allocation; (ii) Banner sets sensor format in parallel with user application.

In particular, Banner employs two key techniques: parallel reconfiguration and format-oblivious memory management. Parallel reconfiguration aims at reconfiguring the sensor while the application is processing frames for the previous resolution such that the reconfiguration latency is hidden. Format-oblivious memory management aims at maintaining a single set of frame buffers – regardless of resolution – to eliminate repeated invocation of expensive memory allocation system calls.

## System Overview

Banner is a media framework that allows applications to request sensor resolution reconfiguration with seamless frame delivery. It exposes a system call to the application and resides in the kernel as a camera host driver to interact with the video buffer driver and the camera driver, as shown in Figure 3.3. Banner minimizes the required number of system calls for vision applications to reconfigure the sensor resolution.

Figure 3.4b depicts the rapid sensor resolution reconfiguration procedure in our proposed Banner framework. When starting the stream, the system sets up the sensor, the sensor host, and the buffer with the highest supported resolution in a sequential procedure, as it does with the V4L2 framework. However, after the application requests a new resolution, Banner will not go through all steps in the sequential procedure again. Instead, Banner maintains the stream without reallocating buffers and then asynchronously reconfigures the sensor in parallel through only one `ioctl` call from the application. Frames at new resolution will be returned after reading out  $N$  frames (determined by the number of buffers requested) already captured with previous resolution. Resolution reconfiguration in Banner is rapid and continuous, i.e., without any frame drop. The procedure for stopping the capture and closing the camera follows the same sequential procedure in V4L2 framework.

### 3.3.1 Parallel Reconfiguration

As we discussed in §3.2, resolution reconfiguration in the current V4L2 framework follows a strict sequential procedure. This sequential reconfiguration procedure introduces both a substantial end-to-end reconfiguration latency and a substantial frame-to-frame reconfiguration latency. In Banner, sensor resolution reconfiguration is completed in parallel while the application is processing frames. By doing so, the

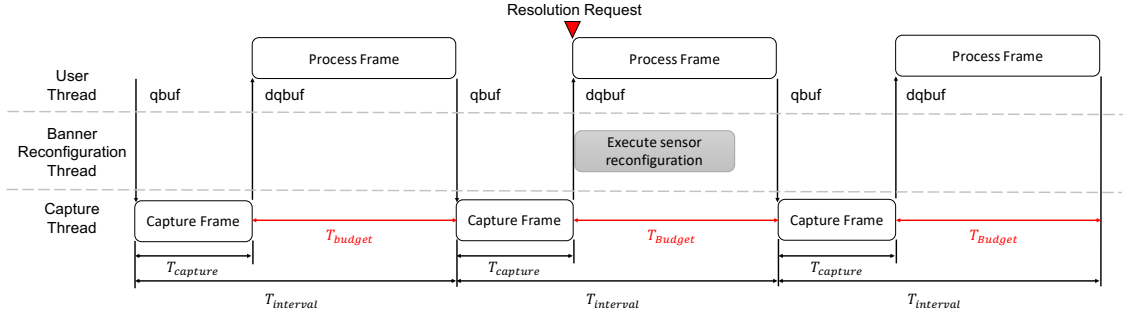


Figure 3.5: Banner reconfigures sensor resolution in parallel with application processing frames in the reconfiguration timing budget (a function of frame interval and capture time) such that reconfiguration latency can be hidden.

frame-to-frame resolution reconfiguration latency is fully hidden.

To achieve this, the parallel reconfiguration module is designed based on three considerations. First, the sensor is not always busy; there is an idle time between captures. Second, the reconfiguration thread cannot be interrupted, otherwise the end-to-end latency will be increased. Dequeuing a buffer signals that a capture is complete and queuing a buffer signals the next capture. The system should identify the right time to reconfigure sensor. Third, reconfiguration itself takes time, due to camera driver implementations and camera hardware limitations.

To resolve these considerations, *thread-level concurrency* can address the first and second considerations, while *a reconfiguration timing budget* can address the second and third considerations. Altogether, Banner can schedule the right time to reconfigure the sensor and trigger the next capture. The parallel reconfiguration strategy is illustrated in Figure 3.5.

## Thread-Level Concurrency

The crux of the parallel reconfiguration is to utilize thread-level concurrency to reconfigure sensor resolution. In the current V4L2 framework, in addition to a main thread, there is a *capture thread* responsible for capturing frames. This capture thread is frozen until it is woken up by the application queuing a buffer for frame capture. The capture thread and the main application thread process in parallel. Although the sensor is busy capturing frames when the capture thread is awake, it is free for reconfiguration while the capture thread is frozen. For Banner, we design a *reconfiguration thread* that can work in parallel with the application thread. This thread processes reconfiguration requests while the application processes frames and the capture thread is frozen. We choose to create a reconfiguration thread, considering the latency penalty incurred by waking up the capture thread. Reconfiguration thread and the main thread are joined before they wake up the capture thread for the next capture. Banner uses atomic read/write to ensure thread safety.

## Reconfiguration Timing Budget

The Banner reconfiguration thread cannot reconfigure the sensor when the capture thread is active. Therefore, a resolution reconfiguration timing budget needs to be defined for the reconfiguration thread to work with. We define resolution reconfiguration timing budget  $T_{budget}$  in Equation 3.1 as a function of frame interval  $T_{interval}$  and capture time  $T_{capture}$ .  $T_{interval}$  – the interval between consecutive frame captures – is defined by the application as the interval between two consecutive `ioctl(VIDIOC_QBUF)` calls from the application.  $T_{interval}$  is typically held stable to ensure good user experience.  $T_{capture}$  varies from frame to frame, influenced by the capture parameters such as the exposure time and resolution.  $T_{budget}$  is equal to frame interval  $T_{interval}$  minus

the required capture time  $T_{capture}$ , i.e.,

$$T_{budget} = T_{interval} - T_{capture} \quad (3.1)$$

It is important to ensure that sensor resolution reconfiguration is finished in the reconfiguration timing budget such that the reconfiguration thread is not interrupted by the wake up of the capture thread. Otherwise, capture at the new resolution will be delayed by another capture with the old resolution, which causes both end-to-end and frame-to-frame reconfiguration latency to be unpredictable.

In our implementation, we have observed that the reconfiguration timing budget is long enough that the reconfiguration latency can be completely hidden by the main application thread. That is, the frame-to-frame latency is eliminated. Seen from the application side, the frame rate is stable in Banner even between two frames at different resolutions. Based on our evaluation, we can maintain more than 30 FPS for an offloading application with only 10 ms reconfiguration timing budget. We note that different image sensors may require different amount of time to reconfigure the resolution.

That being said, theoretically, if an application operates at an unstably fast frame rate, then Banner could potentially delay the delivery of the frame after the resolution request. For example, this would be the case if the application performs a `memcpy` of the frame to a memory location and immediately queues the buffer for a next capture. Still, in this case, Banner would improve the reconfiguration latency over the legacy V4L2 framework, which would delay frame delivery while executing the full reconfiguration procedure – complete with memory allocation.

### 3.3.2 Format-Oblivious Memory Management

As explained in §3.2, the legacy V4L2 framework synchronizes frame buffer resolutions across all of its modules. Buffers are requested and mapped for a determined resolution before the camera can even start capturing. If the application requests another sensor resolution, the legacy V4L2 framework stops current streams, releases previous frame buffers, and allocates a fresh set of buffers. Thus, synchronizing the format can be very expensive for the resolution reconfiguration procedure.

We propose a format-oblivious memory management that removes resolution synchronization in the resolution reconfiguration procedure. Format-oblivious memory management reuses previously allocated buffers to store frames with different formats, as shown in Figure 3.6. This technique reduces the end-to-end reconfiguration latency and frame-to-frame reconfiguration latency by avoiding unnecessary system calls.

#### **One-Time Buffer Allocation**

Instead of allocating frame buffers every time the application requests a new resolution, format-oblivious memory management only allocates buffers *once* when initializing the camera. To support all formats, the system can allocate for the highest supported resolution by the camera for reuse for any resolution.

Reusing buffers brings several benefits. First, repeated memory deallocation and allocation (`ioctl(VIDIOC_REQBUFS)` and `mmap()`) for different sensor resolutions are now completely avoided. The system call `mmap()` is very time consuming as we discussed in §3.2. Second, current video streams are not discarded. The system calls for turning on and off the video streams – `ioctl(VIDIOC_STREAMON)` and `ioctl(VIDIOC_STREAMOFF)` respectively – are avoided. Both of them consume tens

of milliseconds. Third, since there is only one format at the receiving end, the system doesn't need to set the output state of camera host driver for reconfiguration.

### **Format-Oblivious Frame Delivery**

Format-oblivious memory management delivers the frame to the application not based on the payload calculated by the sensor format, but based on how many bytes are used. When the application requests another resolution, Banner passes the format information to the camera driver and the camera host driver appropriately. As the system needs to maintain the current pipeline of frames, there will be a resolution discrepancy among the frames already captured and the frames to be captured in the new configuration.

Banner solves this problem easily by delivering the frames according to how many bytes are used. Banner and the frame itself will provide enough format information for the application to interpret frames. We argue that as long as the application and sensor know the format at the appropriate times, the frame can be correctly captured, delivered, and interpreted.

Format-oblivious memory management can be realized without any modification to the video buffer driver. The only potential limitation of this approach is that it allocates more memory than needed for low resolution configurations. For example, when configured for 480p resolutions, the frame buffer will occupy the memory footprint of a 1080p frame buffer (6 MB for 3 frames). However, on modern mobile systems, we do not foresee this as particularly problematic; most modern phones have at least 1 GB of RAM. More importantly, the additional buffer allocation does not increase system power, as DDR power consumption is governed by data rate, not allocation. We confirm this in our evaluation.

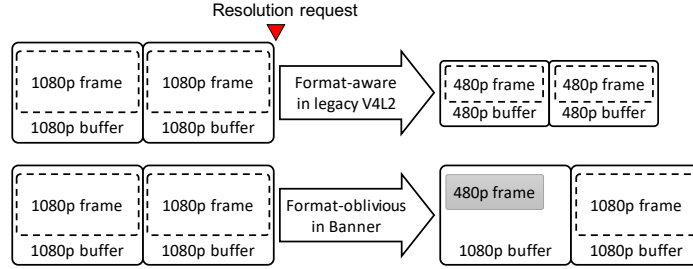


Figure 3.6: After a resolution request, format-oblivious memory management in Banner reuses buffers previously allocated and stores newly configured frames, despite potential format mismatch.

### 3.4 Implementation of Banner

Our Banner prototype is built by modifying the V4L2 media framework in the upstream Nvidia Tegra TX2 Linux kernel 4.4, L4T 28.2.1. In this implementation, **the application reconfigures sensor resolution rapidly through only one `ioctl` system call.**

#### 3.4.1 Parallel Reconfiguration

The goal of Banner’s reconfiguration policy is to utilize idle time in kernel space to change the format of an image sensor. After capturing and processing a frame, the kernel camera host driver returns to an idle state until the next capture. Knowing that the kernel is idle, Banner can use this time to send commands that change the sensor’s format and perform any state changing on the camera host driver side. This sufficiently performs the operations of reconfiguring the sensor resolution. Resolution reconfiguration is initialized by an `ioctl(VIDIOC_RECONFIGURE)` call, from the application which will set a sensor resolution format that is passed from user space to the camera host driver object. This system call will then immediately spawn a kernel thread to perform the reconfigure operation. We spawn a single thread to perform



our reconfiguration as the overhead of spawning multiple times for more parallelism made reconfiguration slower overall. Setting camera host driver states is an immediate operation. The only part of the reconfiguration process that takes significant time is configuring the camera hardware.

### **Configure Sensor Device**

The sensor configuration call changes the state of the camera device. The camera driver module then controls the image sensor directly by making  $I^2C$  or other bus calls. The time that configuring the sensor takes will vary from sensor to sensor as each sensor will have a different protocol for setting sensor format.

### **Update Camera Host Driver State**

Updating the camera host driver's state will prepare it to capture frames at a new resolution. The camera host driver state must be updated immediately after the sensor is reconfigured, as the next captured frame will be at the sensor's new resolution. If it is not done, the next frame will be returned with the old resolution and be interpreted improperly at the application level. The next `ioctl(VIDIOC_QBUF)` operation will use the settings set here to capture a frame. This will also set the input for the frame size of buffers as well as the values required to calculate the size of buffer, so that the application knows how many bytes to read for the frame.

#### *3.4.2 Format-Oblivious Memory Management*

An important optimization in Banner is to reuse memory buffers, as making `mmap()` and `munmap()` calls can take tens of milliseconds based on the frame size. When initializing the device, after calling `ioctl(VIDIOC_REQBUFS)`, the buffers returned should be allocated to the maximum size that will be used by the application.

While reusing the buffers does consume extra memory when the frame size is smaller than maximum, it allows Banner to save reconfiguration latency; the `mmap()` and `munmap()` process does not need to be repeated.

`mmap()` allocates shared memory between the camera device and the application level. Shared memory allows the camera device driver to write frames into the buffer and the application to read from the same address in memory. The shared memory will contain information about the bytes used inside of the buffer, the state (if the buffer is ready to be read from the application level), and the raw frame data. The user application will use the buffer state to know the length of bytes to read out into its own buffer.

### 3.4.3 *User Application Library*

Banner exposes the sensor resolution reconfiguration API to the user application as a V4L2 system call. User applications can call the Banner API, just as they use V4L2 to start video streaming. We use the V4L2 capture example provided in The kernel development community (2021) as a code base for our testing. The example code opens the camera device and initializes all memory needed for capture per the V4L2 specification. The example code then starts a capture loop that will run until a frame count has passed. This capture loop uses the `select` system call to wait until the video buffer driver signals that the buffers are ready for reading. The application takes ownership of the buffer by calling `ioctl(VIDIOC_DQBUF)` and then copies the shared memory to an application buffer before returning it with `ioctl(VIDIOC_QBUF)`.

Our modifications to the example code were minimal.

1. When the application initializes the camera, it counts the number of frame buffers allocated. This count is saved for future reference, as it is equal to the number of frames in any given pipeline.

2. Immediately after a `select`, on any frame, the application can reconfigure sensor resolution by calling `ioctl(VIDIOC_RECONFIGURE)`.
3. After a reconfiguration call, the application starts counting frames returned in the main-loop until the captured frames at previous resolution are read out; at this point, the application's resolution is reconfigured to the new resolution.
4. From this frame onward the frames returned by the driver will be the new resolution.

## OpenCV Hook

When working with Banner in OpenCV, we take our raw frames from the V4L2 capture. OpenCV requires frames to be in BGR format, but the V4L2 camera returns UYVY. To convert frames into a format that OpenCV can manipulate, we use a modified CUDA function from Nvidia. The function originally converted YUYV to RGB, but we manipulated it to convert UYVY to BGR by reordering the image input and output planes. Once we have our BGR frame, it is a 1 dimensional array and still not in a form for OpenCV to work with. To fix this we call the constructor for `Mat`, OpenCV's basic object for handling images. We take care to use the correct parameters for resolution, pixel size, plane count, and our frame data. From there, we can use any OpenCV function to operate on the image, such as `resize`, `imshow`, and `BFMatcher`.

## 3.5 Evaluation

We evaluate Banner within the V4L2 framework on an Nvidia Jetson TX2 board with an ON Semiconductor AR0330 sensor. This Jetson TX2 board has a Quad ARM A57 processor. It is one of the most popular embedded computing devices.

The evaluation answers three questions: *(i)* How much reconfiguration latency did Banner reduce when reconfiguring sensor resolution? *(ii)* How much power efficiency can be gained by reconfiguring sensor resolution dynamically and rapidly with Banner? *(iii)* What does fast sensor resolution reconfiguration mean to computer vision applications?

### 3.5.1 Evaluation Methodology

#### Workloads

To evaluate and validate the effectiveness of Banner for reconfiguring sensor resolution in a variety of vision tasks, we choose three applications integrated with OpenCV. The first application only displays frames at 25 FPS. This application gives us preliminary results of the effectiveness of Banner. The second application offloads frames to a desktop server through a direct connection at 15 FPS. This application demonstrates Banner’s usage in cloud-based vision applications. The third application implements the same marker-based pose estimation as we described in §3.1.1, running at 15 FPS. It verifies that Banner is effective for our target application (Augmented Reality).

All three applications cycle through a set of supported resolutions: 1920x1080, 1280x720, and 640x480. To compare Banner reconfiguration against computational downsampling, we use OpenCV `resize()` function to downscale 1080p frames to 480p (represented as 1080p↓480p). The frame rate is set to be constant across different resolutions in all applications, bounded by the frame rate in the highest resolution, with the help from dynamic CPU and GPU clock scaling.

We measure latency and power. As we defined in §3.1, resolution reconfiguration latency includes *end-to-end reconfiguration latency* which describes how long it takes for the application to receive the first new frame after a resolution is requested,

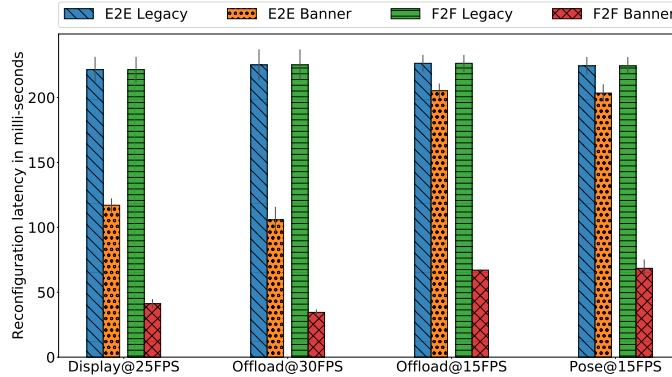


Figure 3.7: Banner reduces end-to-end (E2E) resolution reconfiguration latency and removes frame-to-frame (F2F) latency in all three workloads comparing with legacy in V4L2 framework.

and *frame-to-frame latency* which indicates the interval during which an application receiving no frames after a resolution is requested. Both latencies are measured by retrieving system timestamps at the application level. In each application, they are measured and averaged across 99 samples, i.e., 99 resolution reconfigurations. *Power consumption* is monitored by retrieving the power rail system files on the Jetson TX2 board. These files include SYS\_SoC which monitors the power consumption of the main Tegra core, SYS\_DDR which monitors the power consumption of the LPDDR4, SYS\_CPU which monitors the power consumption of the ARM processor, and SYS\_GPU which monitors the power consumption of the Pascal GPU. Power rail system files are written automatically by the system at 400 KHz. In our evaluation, we measured the power after the application ran at a steady state, in order to minimize the variance. In each measurement, we acquired and averaged 600 readings for each power rail system file.

### 3.5.2 Resolution Reconfiguration Latency Reduction

With parallel reconfiguration and format-oblivious memory management, **Banner completely eliminates the frame-to-frame latency in all three workloads and is able to halve the end-to-end reconfiguration latency**, as shown in Figure 3.7.

In the display workload (at 25 FPS), the average end-to-end reconfiguration latency is reduced by 47% (from 222 ms to 117 ms) and the average frame-to-frame latency is reduced by 82% (from 222 ms to 41 ms). In the slower offloading workload (at 15 FPS), the average end-to-end reconfiguration latency is reduced by 9% (from 226 ms to 205 ms) and the average frame-to-frame latency is reduced by 70% (from 226 ms to 67 ms). In the pose estimation workload (at 15 FPS), the average end-to-end reconfiguration latency is reduced by 10% (from 225 ms to 203 ms) and the average frame-to-frame latency is reduced by 70% (from 225 ms to 67 ms). In addition, in the faster offloading sub-workload (at 30 FPS), the average end-to-end reconfiguration latency is reduced by 54% (from 226 ms to 105 ms) and the average frame-to-frame latency is reduced by 85% (from 226 ms to 34 ms).

We have several observations from these results. First, end-to-end reconfiguration latency and frame-to-frame latency are equivalent in the legacy V4L2 framework. This is because the frames stored in the capture queue are abandoned once there is a new resolution request. If those frames need to be read out and processed before the start of resolution reconfiguration – as they are in the Android OS Hu *et al.* (2018) – the end-to-end reconfiguration latency can be even larger.

Second, the average end-to-end reconfiguration latency and frame-to-frame the latency in legacy V4L2 framework are stable across all three workloads because they all go through the same procedure, though they still have larger standard deviation

compared to Banner.

Third, end-to-end reconfiguration latency and frame-to-frame latency are predictable in Banner because they depend on the frame rate. In Banner, the first frame at new resolution will be received after  $N$  frame intervals, where  $N$  is the number of frames already captured and stored in the buffer queue for the previous resolution. In our case, there are three buffers ( $N$ ) requested and thus, three captures are stored in the buffer queue whenever a new resolution is requested by the application. Therefore, the end-to-end reconfiguration latency in Banner is around three frame intervals. These intervals cause the end-to-end reconfiguration latency to be around 120 ms in Display@25FPS and 200 ms in Pose@15FPS, as shown in Figure 3.7. Frame-to-frame latency in Banner is equal to the inverse of the processing frame rate. The application will receive continuous frames at the same frame rate without noticing the resolution reconfiguration procedure. In other words, Banner eliminates frame drops.

### 3.5.3 Power Efficiency Improvement

Table 3.2 demonstrates that **rapid sensor resolution reconfiguration with Banner enables a substantial power efficiency improvement**. We note the following observations.

First, the resolution-based power efficiency improvement is generic to vision tasks and components across the system (e.g., SoC, DDR, CPU, GPU). In all three evaluated workloads, the choice of sensor resolution influences the power consumption of all stages in the image processing pipeline, including data movement, storage, and processing. The results in Table 3.2 show that SoC, GPU, DDR, and CPU all benefit from processing lower resolution frames in terms of power savings.

Second, the power efficiency improvement is substantial as the sensor resolution drops. In legacy V4L2, the combined power consumption is reduced by 62%, 60%,

Workload	Resolution@FPS	Legacy V4L2/Banner				
		SoC	GPU	DDR	CPU	Total
Display-only	1920x1080@25	1149/1149	910/918	1869/1836	2460/2190	6388/6096
	1280x720@25	1073/1073	611/559	1727/1704	1076/1100	4487/4436
	640x480@25	617/669	306/306	826/860	691/681	2440/2516
	1080p↓480p@25	1078/N/A	613/N/A	1690/N/A	1035/N/A	4416/N/A
Offloading	1920x1080@15	1073/1146	536/544	1629/1638	1967/2027	5205/5355
	1280x720@15	688/700	230/230	863/856	617/630	2398/2416
	640x480@15	617/630	230/230	702/687	540/554	2089/2101
	1080p↓480p@15	1073/N/A	382/N/A	1606/N/A	1032/N/A	4093/N/A
Pose estimation	1920x1080@15	1281/1149	1701/1448	1940/1875	2524/2527	7446/6999
	1280x720@15	1230/1225	1058/987	1814/1795	1271/1203	5373/5210
	640x480@15	1210/1150	589/540	1635/1637	928/916	4362/4243
	1080p↓480p@15	1227/N/A	850/N/A	1727/N/A	1260/N/A	5064/N/A

Table 3.2: Total system power consumption (mW) is reduced by 62%, 60%, and 42% as sensor resolution is reduced from 1080p (1920x1080 in legacy V4L2 ) to 480p (640x480 in legacy V4L2), in each workload accordingly. In addition, physically reconfiguring sensor resolution to 480p (640x480 in Banner) consumes 43%, 49%, and 16% less total system power than downsampling 1080p to 480p (1080p↓480p in legacy V4L2), in each workload accordingly.

and 42% as sensor resolution is reduced from 1080p to 480p in display, offload, and pose estimation workloads respectively. Thus, the power efficiency of a mobile vision task can be significantly improved if the system allows dynamic sensor reconfiguration when it can sacrifice resolution.

Third, reconfiguring sensor resolution physically is much more power efficient than other alternatives, i.e., computational downsampling. 1080p↓480p@FPS in Table 3.2 shows the power consumption of downsampling 1080p frames to 480p in the legacy V4L2 framework. Comparing with reconfiguring sensor resolution physically to 480p in Banner, downsampling consumes 43%, 49%, and 16% more power in display, of-



load, and pose estimation accordingly.

### **Effectiveness of Dynamic Reconfiguration**

To demonstrate the power efficiency improvement brought by Banner for reconfiguring sensor resolution dynamically, we conduct a simple experiment in which the sensor resolution cycles through 1080p, 720p, and 480p every 10 frames (a randomly chosen number) in a total of 1000 frames. This results in 99 resolution reconfigurations. We run this pattern with our CPU-based cloud-based offloading workload working at 30 FPS. The results in Table 3.3 show that even in legacy V4L2 framework, reconfiguring sensor resolution dynamically (99x-reconf.-V4L2) can reduce 20% of the combined system power consumption comparing with constantly working at 1080p – notably, there are substantial frame drops with each reconfiguration in the legacy V4L2 system.

Meanwhile, reconfiguring sensor resolution with Banner (99x-reconf.-Banner) can further reduce total power consumption by 9% and without the frame drop penalty, compared with 99x-reconf.-V4L2. These power savings come from the use of fewer operations to reconfigure the sensor format and no repeated memory allocation in Banner. The power consumption of 99x-reconf.-Banner is roughly about the same as constantly working at 480p.

### **Power Overhead of Banner**

As shown in Table 3.2, comparing between Banner and legacy V4L2 framework, there is no obvious power overhead. Specifically, Banner does not consume more DDR power despite its allocation of more memory than the active resolution requires. This is because DDR power consumption is based on data rate, not buffer size Micron Technology (2019).

Resolution-Framework	SoC	GPU	DDR	CPU	Total
1920x1080-V4L2	803	230	951	879	2863
1280x720-V4L2	637	230	686	693	2246
640x480-V4L2	612	230	618	613	2073
99x-reconf.-V4L2	659	230	719	691	2299
99x-reconf.-Banner	620	230	644	600	2094

Table 3.3: Reconfiguring sensor resolution dynamically (99x-reconf.-Banner) can reduce 27% of the combined system power consumption (mW) comparing with constantly working at 1080p (1920x1080-V4L2), measured with our CPU-based cloud-based offloading workload working at 30 FPS.

### 3.5.4 Implications

Banner enables rapid sensor resolution reconfiguration. This unlocks a more than 49% system power consumption reduction by reconfiguring the image sensor resolution from 1080p to 480p comparing with computationally downsampling to 1080p $\downarrow$ 480p. As we mentioned in our motivation, in a variety of vision tasks, the image resolution needs to be configured dynamically to adapt to environmental changes in order to maximize the power efficiency. For example, the required sensor resolution can be determined dynamically based on the continuously changing distance between the image sensor and the marker in a marker-based pose estimation application. Our evaluation in the marker-based pose estimation application on the Jetson/AR0330 system reveals that the estimated pose accuracy can be maintained ( $\pm 0.1$  cm MSE translation vector error) even if the image resolution is reconfigured from 1080p to 720p and then to 480p as the distance between the image sensor and the marker is reduced from 40 cm to 20 cm. This results in a 28% power consumption reduction between 1080p and 720p and a 42% power consumption reduction between 1080p and 480p.

### SPLIT-PROCESS APPLICATION DEVELOPMENT FRAMEWORK

Augmented Reality (AR) enables smartphone users to interact with virtual content spatially overlaid on a continuously captured physical world. Under the current permission enforcement model in popular operating systems, AR apps are given Internet permission at installation time, and request camera permission and external storage write permission at runtime through a user’s approval. With these permissions granted, any Internet-enabled AR app could silently collect camera frames and derived visual information for malicious intent without a user’s awareness. This raises serious concerns about the disclosure of private user data in their living environments.

To give users more control over application usage of their camera frames and the information derived from them, we introduce LensCap, a split-process app design framework, in which the app is split into a camera-handling visual process and a connectivity-handling network process. At runtime, LensCap manages secured communications between split processes, enacting fine-grained data usage monitoring. LensCap also allows both processes to present interactive user interfaces. With LensCap, users can decide what forms of visual data can be transmitted to the network, while still allowing visual data to be used for AR purposes on device. We prototype LensCap as an Android library and demonstrate its usability as a plugin in Unreal Engine. Performance evaluation results on five AR apps confirm that visual privacy can be preserved with an insignificant latency penalty ( $< 1.3$  ms) at 60 FPS.

## 4.1 Introduction

Augmented Reality (AR) provides a unique interactive experience of virtual objects overlaying on top of the real-world environment enhanced by continuous capturing, processing, and rendering of visual data through a mobile device. The development of mobile devices and AR frameworks have enabled applications of AR in many fields, including education, entertainment, medicine, navigation, shopping, etc., and the future of AR market is expected to continue its rapid growth FACEBOOK (2021); Ashe (2017); Hu *et al.* (2018); Shaikh *et al.* (2019); Hu *et al.* (2019); Paine (2020).

Unfortunately, running AR apps on today’s mobile devices poses serious privacy concerns, potentially revealing private user information in a user’s visual environment to third party entities without the user’s knowledge. Under the current permission enforcement model, an AR app is given Internet permission at installation time and granted camera permission and external storage write permission at runtime by users. Developers are required to prompt users with contextual information about why certain permissions are required, but such permissions are seemingly justified for proper AR operation; camera frames are necessary to visually integrate virtual content with a user’s physical environment and Internet connectivity is needed for cloud-powered services or multiplayer networking. But once enabled, malicious developers of AR apps could silently collect camera frames and the information derived from them for malicious intent, including sending visual data to a private server, unbeknownst to the user. Without granular control over what kind of visual data is accessible for local storage or cloud storage, those collected camera frames could contain very private data at any given time, ranging from credit cards left on the table, text recognized from business documents on laptop monitors, to critical facial identities. **How do we protect users from surreptitious collection of visual data while maintaining**

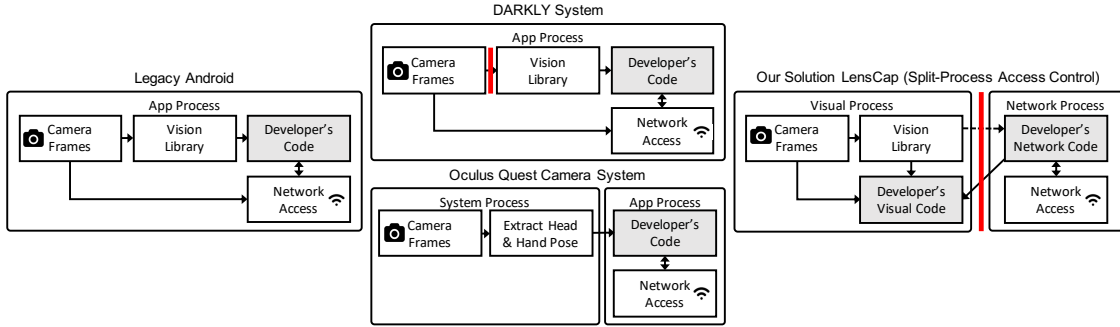


Figure 4.1: In legacy Android, AR developers could collect any camera frames and information derived from them without user awareness. Related solutions in DARKLY Jana *et al.* (2013) and Oculus Quest system Facebook (2021) only allow developer access to pre-defined processed visual data, and do not allow rendering of the camera frame. LensCap adopts split-process access control to allow developer’s code to freely access frames for AR rendering while managing what visual data is accessible to the network.

### usable visual computing for AR applications?

Related solutions attempt to protect visual privacy by processing camera frames into privacy-preserving visual features and only give apps access to those features, or a region of the camera frame defined by the users Jana *et al.* (2013); Roesner *et al.* (2014); Aditya *et al.* (2016); Olejnik *et al.* (2017); Raval *et al.* (2014, 2016), as shown in Figure 4.1. However, this is too restrictive for AR apps, which need the ability to visually render the entire frame to provide the camera view as a backdrop for virtual AR overlays. Information flow control protects sensitive data through dataflow analysis and taint tracking Enck *et al.* (2010); Arzt *et al.* (2014); Fernandes *et al.* (2016); Wang *et al.* (2019). However, most information flow control works are only effective on data types involving low throughput. TaintDroid introduces 500 ms latency capturing still pictures. FlowFence consumes 100 ms processing 612x816

camera frames for face recognition in security camera apps. This latency overhead is untenable for AR apps, which must process at 16.6 ms per frame to maintain 60 FPS. Compartmentalization attempts to partition an app to confine private data in a secured hardware or software environment Herbster *et al.* (2016); Costan and Devadas (2016); Raval *et al.* (2019); Dawoud and Bugiel (2019); Track and Kilpatrick (2003); Huang (2020), usually against threats from external third-party plugins or advertisement libraries.

To address the privacy disclosure of continuous camera usage, we introduce LensCap, an application development framework built on top of split-process access control Jensen *et al.* (2019), which allows users with fine-grained and proactive control over the app’s potential transmission of camera frames and the information derived from them. The idea of split-process access control is not new; Android OS splits the `mediaserver` process into multiple processes to restrict their usage (after v7.0 Android Developer (2021)). In LensCap, the split-process paradigm is adopted in the application layer, which is integrated into the app development flow. An AR application is split into a *visual process* with full access to operate on camera frames (but with network permission revoked) and a *network process* to maintain Internet communications (but with camera permission revoked), enforced by extending the legacy Android permission enforcement. We enable both processes to present user interfaces through screen-based overlay composition. Then, data related to camera frames that need to be used in the network process can only be transmitted out of the visual process boundary through our trusted LensCap communication services, wrapped around trusted AR frameworks, and subject to the user’s monitoring and approval through LensCap data usage notifications at a fine granularity. If users wish to allow network access to entire camera frames, e.g., for social media sharing or cloud-powered vision Simoens *et al.* (2013); Ha *et al.* (2014), they can enable such permission. On the other hand,

if a user wants to limit network access to only the camera pose, e.g., for multiplayer purposes, the user will be able to do so while still enjoying a full AR overlay on the device.

Thus, LensCap split-process app development framework enables: (i) AR apps and vision libraries to have expressive access of camera frames, their processing, and their rendering; (ii) fine-grained user control of the potential transmission of visual data; (iii) detailed context provided to users, regarding what data is sent to the cloud and at what times. We acknowledge that split-process frameworks may still be vulnerable to security threats through covert channel and side channel attacks Lampson (1973); Reardon *et al.* (2019), which are beyond the scope of this paper. However, process-based partitioning of resources narrows the attack surface and could enable protection measures, such as permission plugins Raval *et al.* (2019). Ultimately, LensCap relies on the operating system to protect the communication channels and the app’s internal storage to secure visual privacy across the split-process boundary Android Developers (2021b); Guo (2014).

We prototype LensCap as an Android library that can work with standalone Android projects, as well as with Unreal Engine (UE) projects. In UE, LensCap serves as a plugin, through which a UE game compilation process will automatically generate and compile the split-process Android project structure. The communication channels between split processes in UE are provided to AR developers as Blueprint-callable and leverage the Android environment through the Java Native Interface (JNI). The data usage monitor is implemented as Android notifications, which provides the user with a rendered status of potential visual data collection.

We evaluate LensCap in five cloud-based AR applications that require the sharing of different types of image features, including camera pose, light estimation, point cloud, face region, and the camera frame. We find that the interactive latency between

split processes and the overhead in app performance is negligible, even at 60 frames per second. Our user study further validates the performance similarity from the user’s perspective and the improvement in user confidence while using untrusted AR apps.

## 4.2 Background

In this paper, we study the AR development flow in UE, as well as the permission control model and security enforcement in Android OS. They are similar across other game development platforms and operating systems such as Unity for iOS.

### 4.2.1 Mobile AR Development

Powerful real-time 3D creation tools such as Unreal Engine Epic Games (2021) have gained popularity in creating cutting-edge content in immersive interactive experiences. The basic building block in UE is the module. Each module exposes itself to other modules through a public interface. Developers can include a set of modules to realize desired app functionalities. For example, to develop AR apps relying on the ARCore library, app developers need to declare the `AugmentedReality` module and the `GoogleARCoreBase` module in dependencies. Apart from those standard modules, developers can create plugins to add per-project code and data to extend runtime gameplay functionality of the app.

UE supports all popular mobile platforms including Android and iOS. To build and run UE apps in the Android environment, UE creates an intermediate Android project based on two workflows. First, UE provides all source files that are necessary for the intermediate Android project to be compiled and run, such as a `GameActivity.java` template, on top of which developers can customize logic and functionalities. Second, all necessary assets for developing the UE project such as the level Blueprint are



compiled into a `.so` library as the native code. The compiled Android application interacts with UE features through JNI.

#### 4.2.2 *Permission Control*

Android requires apps to define permissions in a signed manifest file to manage the security of various operations Android Developers (2021a). Permissions are categorized into different protection levels as normal, signature, and dangerous, in which dangerous permissions must be prompt to and further granted by users. Permissions for Internet (`permission.INTERNET`), camera (`permission.CAMERA`), and external storage write (`permission.WRITE_EXTERNAL_STORAGE`) are essential to an AR app.

Screen buffer capture (screen reading/recording) is also governed by Android signature permissions. Android apps can specifically prompt the user when screen buffer capture is required. Users can enable this at their own discretion, understanding that everything on the screen will be accessible by the application, including any visible camera feeds.

Internet permission is categorized under the normal protection level. It is automatically granted at installation time by the system and users will not be notified that the permission is granted. Camera permission is essential for protecting user’s visual privacy, and is therefore categorized under the dangerous protection level. Users will be notified to grant the camera permission in a prompt dialog. In Android 11, users are able to grant one-time permission to application’s camera usage called “Only this time”. However, the app will still have continuous camera access during that one-time. Write external storage permission is also categorized under the dangerous protection level which requires the user approval at runtime.

### 4.2.3 Security Enforcement

In Android, security is enforced through app sandboxing. Each app runs in a separate sandbox with a unique application identity (UID) given during installation. Sandboxing ensures each app has its own process and data storage associated with the UID. App sandboxes cannot interact with each other and only have limited access to system services by default. If a certain permission is granted, it will be reflected in the context of the application package and UID. Android conducts a permission check if the app sandbox requests access to specific system services (e.g. camera service) or resources belonging to other apps.

Sandboxing introduces the need of inter-process communication (IPC). Android implements Binder IPC as an essential mechanism to perform operations between processes, such as passing messages and requesting system services. Binder IPC provides functionalities to bind to functions and data between different execution environments. The Binder IPC driver is implemented in the kernel. It exposes basic kernel-understood functions to the application developers through the IBinder interface at the framework layer defined using the Android Interface Definition Language (AIDL).

## 4.3 Threat and Trust Model

### **Threat Model**

We focus on scenarios involving third-party AR apps that require Internet connectivity. Relying on cloud or edge-based computing platforms empowers mobile AR apps with additional computing power and dynamic access to networked resources, e.g., game state, object models/textures, content updates, etc. This model also includes collaborative multi-user AR games in which different AR users could share

information such as camera states, point clouds, and lighting estimations for more accurate tracking and rendering. However, while they provide useful and engaging functionality, **we assume that all such third-party AR apps cannot be fully trusted.** Privacy leakage through camera frames could happen at any time during the AR experience, with apps surreptitiously collecting visual data without user awareness. The visual data may be full camera frames themselves, derived semantic information (e.g., text or face identities), or compressed representations. The data could be immediately transmitted upon capture or stored locally before sending the data over the network, e.g., bundled with other data upload.

### **Trust Model**

We make three assumptions in our trust model. *(i)* We assume that operating systems such as Android and iOS are trusted to perform runtime permission check, hardware platforms are secured against attacks, and official AR frameworks such as Google ARCore and Apple ARKit are trusted to operate on camera frames for AR functionalities. These components are usually secured through a set of operating systems and hardware security measures Android Developers (2021b); Guo (2014). *(ii)* We assume that visual data sharing is valid as long as users are aware of it and specifically grant it. That is, AR applications could be allowed to share camera frames or information derived from them with the user’s approval. *(iii)* We assume that data downloaded from the Internet or read from the memory is not tampered with. The protection of AR visual output is actively discussed in other research works Lebeck *et al.* (2018, 2017); Ringer *et al.* (2016).

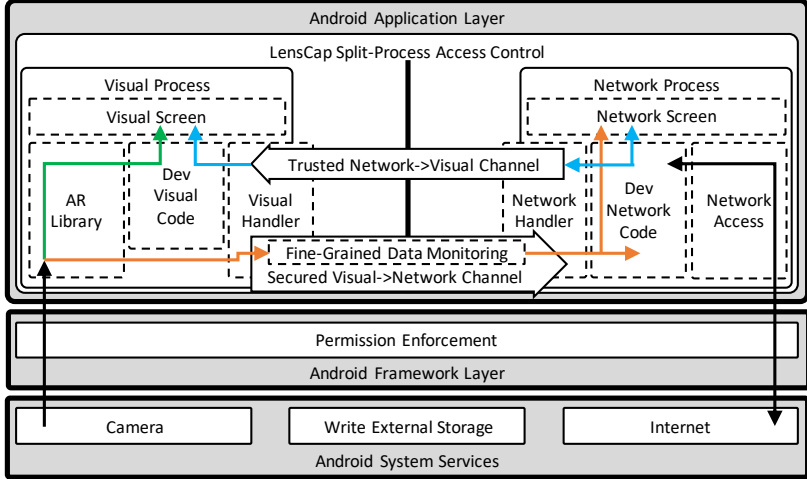


Figure 4.2: LensCap in Android. AR library output can go to developer’s visual code to render on visual screen (green), or to network code through secured LensCap Visual→Network channel (orange). The blue arrow shows Network→Visual dataflow, defined in §4.4.2.

### Challenges

There are many challenges to protect visual privacy in real-life. *(i)* The AR experience must be quick to respond to user movement and interaction; the system solution should not contribute any visible performance overhead. *(ii)* Visual details need to be protected without reducing the amount of information that an AR application requires. *(iii)* Apps may require Internet connection to utilize more powerful cloud- and edge-based computing and/or maintain state on networked resources. *(iv)* There is a trust gap to be mitigated between users and AR apps in terms of the data claimed to be collected and the data actually collected.

## 4.4 Design of LensCap

We propose LensCap, a framework that secures visual privacy in AR apps through *(i)* enforced split-process access control, *(ii)* secured communication channels for pro-

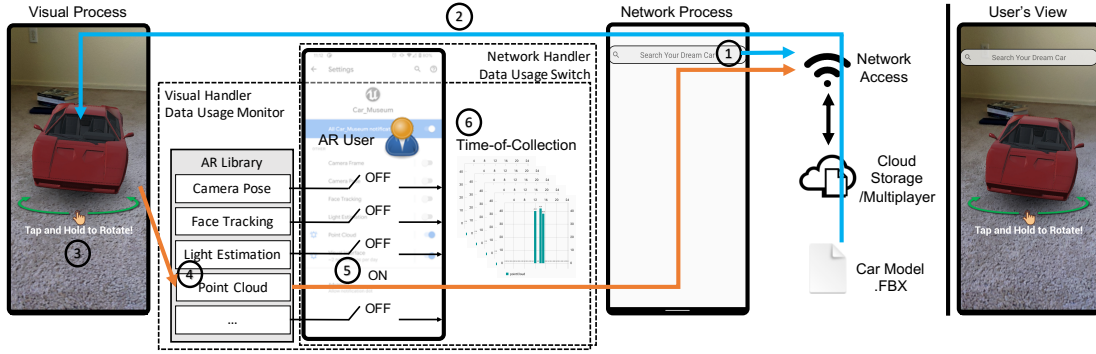


Figure 4.3: ① A user searches for a car in a search bar presented through the network process screen. ② The corresponding car model is downloaded from the cloud, transmitted to the visual process through LensCap Network→Visual channel, and rendered on the detected AR tracking plane in the visual process. ③ The user can spatially interact with the virtual 3D car model. Meanwhile, ④ point cloud positioning data is shared over the network for shared tracking for a multiplayer AR experience, subject to ⑤ user’s approval secured by LensCap data usage monitor and switch in Visual→Network channel. ⑥ Visual data usage can be reviewed by the user.

cesses to securely exchange data, (iii) screen-based overlay composition, and (iv) fine-grained data monitoring. Figure 4.2 shows LensCap in the Android system.

#### 4.4.1 Enforced Split-Process Access Control

LensCap requires applications to be split into a visual process and a network process. Through this process-based partitioning, the permission enforcement in the operating system can assure that AR apps isolate visual processing from network access except via explicit user approval.

*The visual process* is responsible for processing and rendering camera frames, and thus is allowed to operate on them directly. Expressive functional use of the frames

and visual data allows developers to program computer vision and image processing operations freely, as long as the features derived from those camera frames do not leave the visual process boundary, except with explicit user approval.

*The network process* is not allowed to operate on camera frames or other visual data, except with explicit user approval. However, the network process is critical to edge- or cloud-based AR apps for its capability of providing Internet communications, as well as writing to the external storage. LensCap sequesters the network process from direct access to the camera frames by revoking camera permissions, but offers it upload and download access to the network and write access to external storage.

*The permission enforcement*, residing inside the Android runtime framework, verifies that when a process attempts to open the camera, it does not have either the network permission or write external storage permission granted. LensCap also verifies that when a process attempts to write to the external storage, it does not have the camera permission granted. Violations will throw exceptions to notify users about potentially malicious behavior.

#### 4.4.2 Secured Communication Channels between Split Processes

To securely support a range of operations between the visual process and the network process, LensCap governs communication between the two processes through two data *Handlers*, one for each process, which enact two channels: *Network*→*Visual* and *Visual*→*Network*.

**Network**→**Visual is implicitly trusted.** Thus, the network process could send data to the visual process freely, as visual privacy would still be confined inside the visual process. The protection of visual rendering is beyond the scope of this work, but studied in related works Lebeck *et al.* (2018, 2017).

**Visual→Network needs to be explicitly secured.** LensCap scrutinizes the data sent from the visual process to the network process and presents access control and access logs to users. Users should assume that any data that travels across the Visual→Network channel are visible to the network. This involves any data that could possibly be used to invade user’s privacy, ranging from visual details as small as the camera pose to data as large as the entire camera frame.

To prevent developers from hiding visual information in computed data, LensCap’s Visual→Network channel can only transmit specific untainted visual data, including camera frames or direct outputs from the AR library, e.g., camera pose and face tracking features. LensCap restricts all other forms of transmission on the Visual→Network channel. In §4.4.4, we go into further detail into how such visual data access is monitored by the system, presented to the user, and selectively permitted by the user.

#### 4.4.3 *Screen-Based Overlay Composition*

The screen-based overlay composition allows developers to expressively create screen-based interactions through the visual process screen and the network process screen. Both screens include support for the full array of touch-based user interfaces: buttons, sliders, swipes, or custom-designed screen-based interactions. LensCap composes the visual output by overlaying the network process screen surface over the visual process screen surface.

Figure 4.3 shows an example in which the tap-hold-rotate behavior to interact with the virtual object is implemented in the visual process and rendered in the visual process screen, as it would be in a legacy app. Meanwhile, an interactive search bar is hosted in the network process screen, through which the user can search for different 3D models and download them for rendering.

#### 4.4.4 Fine-Grained Data Monitoring

LensCap monitors the usage of visual data with a *data usage monitor* and a *data usage switch*, as shown in Figure 4.3.

##### The Data Usage Monitor

The data usage monitor wraps around the vision library API for three purposes: *(i)* it lets app developers utilize the AR library and the Visual→Network communication channel; *(ii)* it checks the user permission and then documents when and how often each monitored function is called; *(iii)* it ensures data computed from the trusted vision library and the data sent through the visual data handler are identical, i.e., untainted.

For *(i)*, LensCap wraps around the AR library, preserving original usability. The developer’s app invokes the LensCap AR functions to obtain original AR library function output. The developer’s app invokes LensCap transmission functions to request the sending of AR library output or camera frames to the network process. For *(ii)*, LensCap monitors each wrapper function with a separate permission label, counter, and timer. LensCap updates the permission label according to the user’s choice in the data usage switch. Upon visual data transmission across the Visual→Network channel, LensCap increments the counter and timer, documenting the time of visual data access from the network process. For *(iii)*, LensCap acquires a copy of the data when each monitored wrapper function is invoked by the app. The visual data is checked for identical comparison with the copy before transmitted through the Visual→Network channel. We have found that memory comparison (e.g., `memcmp`) is sufficiently performant and hash comparisons are not needed.



## The Data Usage Switch

We design the data usage switch to present data usage notifications and logs to users with two considerations. First, the data usage switch allows users to customize the visual data they allow to be shared at a fine granularity, i.e., users are able to specifically disable the Visual→Network communication for each monitored function to prohibit its output from being passed out of the visual process boundary. Second, the data usage switch collects access timing information from the data usage monitor and presents access charts to users visually. Through this, users are able to transparently see what visual features are potentially shared over the network and at what times.

### 4.5 Programming Model

As with other operating system privacy changes, LensCap requires developers to alter their app development patterns. In LensCap, developers must partition their applications into the visual and network processes and manage communications between the processes. Here, we describe four template scenarios to illustrate the programming model with LensCap split-process access control integrated into an AR development flow, as shown in Figure 4.4.

LensCap allows developers to consider the “principle of least privilege” and specify only the necessary level of access control for each process. In this light, Scenarios 1 and 2 are supported without any special Visual→Network privilege while still enabling immersive interactive AR functionality. Meanwhile, Scenario 3 only requires the user to allow specific visual data, e.g., camera pose, to be shared across the boundary. Finally, Scenario 4 allows developers to share camera frames and visual data over the network with the explicit permission of the user. These scenarios serve as examples

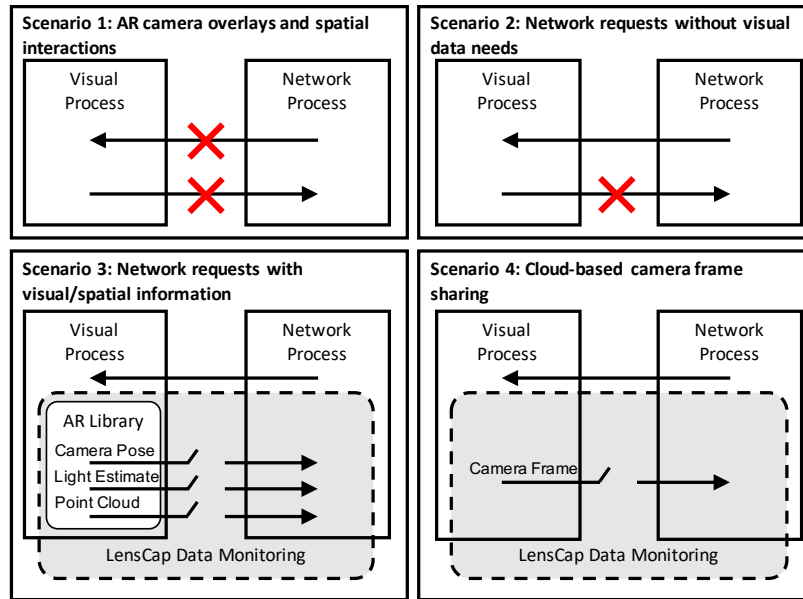


Figure 4.4: LensCap protects AR users in apps where outputs from AR libraries or the entire camera frames are to be sent to the network.

of how applications can be developed in the LensCap environment. Complex applications can be thought of as combinations of these scenarios; Figure 4.3 shows an app that has elements of Scenarios 1 (green), 2 (blue), and 3 (orange). Note that it is developer’s responsibility to take care of the application behavior if the visual data requested are not permitted by users in scenario 3 and 4.

### Scenario 1: AR Camera Overlays and Spatial Interactions.

Developers can program spatial interactions with the AR scene in the visual process, e.g., allowing users to place, spin, scale, and otherwise interact with virtual 3D object content. In this scenario, the application does not need to request any LensCap permissions, as all visual data can be contained in the visual process.

### **Scenario 2: Network Requests Without Visual Data Needs.**

Developers can program network requests in the network process. These requests can be triggered by non-AR on-screen canvas user interface (UI) elements (buttons, sliders, etc.), by time-based events, or by other activities that don't require visual data. Notably, in this scenario, LensCap allows network process to influence activities in the visual process through the unidirectional Network→Visual channel, e.g., specifying what to render based on button UIs that are tapped in the network process. Downloaded data can also influence the spatial rendering, providing object models, data to visualize, image textures, synchronized game state, etc. As in Scenario 1, the application does not need to request any LensCap permissions, as nothing from the visual process needs to interact with the network process.

### **Scenario 3: Network Requests With Visual/Spatial Information.**

Developers may need to share visual or spatial information among multiple devices for multiplayer positioning, joint illumination estimation, or other cloud-based activities. In this case, LensCap allows developers to request user permission to expose specific AR library outputs over the Visual→Network channel. For example, developers can explicitly request the camera pose permission in order to use AR positioning data inferred in the visual process for network-based rendering. Similarly, the developer can explicitly request point cloud permission to enable point cloud sharing for a shared multiplayer AR rendering experience. This scenario also facilitates AR-based spatial interactions that trigger network downloads, as spatial interactions may be inextricable from both visual data and network data. Altogether, in this scenario, LensCap only requires users to enable specific visual data to be shared, without exposing their entire camera frame and the potential secrets or embarrassments therein.

## Scenario 4: Cloud-Based Camera Frame Sharing.

In some apps, users may want to send their entire camera frame to the cloud to enable live-streaming and social media sharing. Other apps may require cloud-based processing of camera frames for resource intensive and/or collective vision operation. In these apps, developers can request users to allow camera frames to pass through in the LensCap Visual→Network channel. For this scenario, users can be made aware that their visual privacy might be leaked and selectively disable camera frame access at their discretion. Furthermore, users can review the LensCap data usage monitor to observe when camera frames are collected to infer malicious intent.

### 4.6 Implementation of LensCap

In this section, we describe a prototype of LensCap in the Android ecosystem. We implement developer support in the form of libraries and automated compilation tools to support both standalone Android development and/or UE development revolving around the ARCore framework. The implementation is generic to other game design platforms, e.g., Unity, and other AR frameworks, e.g., Apple ARKit.

#### 4.6.1 *LensCap Permission Enforcement*

We implement the split-process permission enforcement in the Android framework layer (AOSP v9.0.0\_r46). The implementation involves `CameraManager` in the `Camera2` API, `ContextWrapper` in the `Content` API, and the Android namespace defined in the `xmlns:android`.

LensCap defines the `permission.LENSCAP` manifest permission attribute. Once the app is started, LensCap prompts users to ask if they “allow the app to use LensCap to monitor and validate visual information uploaded to the Internet”. If users choose “ALLOW”, LensCap permission system will be enforced. Inside the `CameraManager`,

LensCap verifies if either Internet or write external storage permission is granted when camera permission is to be granted, relying on the `PackageManager` and `AppGlobals` for retrieving the permission of each app based on its UID. Violations will throw security exceptions to prevent the app from accessing the camera service. The implementation in `ContextWrapper` for protecting the visual privacy from being written to an external storage is similar. The app calls `getExternalFilesDir()` to inquire the absolute path of the directory on the primary shared/external storage device, prompting users to grant write external storage permission. Here, LensCap verifies if the app also has camera permission granted. Violations will lead to `null` returned as the path to invalidate the writing.

#### 4.6.2 *Split-Process for UE Development*

Developers can use LensCap to split their app in the standard Android environment. However, to facilitate game engine-based development, we implement an automated tool to generate LensCap-provisioned Android projects from UE projects as part of the compilation process.

We keep the structure of the UE-generated intermediate Android project and reuse its main application module as either the visual process or the network process. Then (i), for the UE process, we automate the insertion of our LensCap APIs into the main `GameActivity.java` file by adding function definitions into the `<gameActivityClassAdditions>` inside the plugin's XML file. These APIs are described in §4.6.3 for realizing LensCap-verified interactions between the network process and the visual process. In addition, startup permissions related to write external storage and Internet communication given to the visual process are automatically removed by modifying the source `AndroidManifest.xml` file. (ii), for the non-UE process, we implement it as an individual app package which has its own workspace,

from source code to build configurations. To automate the generation of non-UE processes, we provide the source code of the LensCap app package as a third-party library for UE, together with the two data handlers. During compilation, the source code is copied to the build directory of the project and built with the visual process app together. Developers can implement processing logic inside UE as in the legacy development flow and/or utilize the Android process scenario templates LensCap provided to create complex network-visual interactions.

### 4.6.3 Secured Communication Channels

The visual/network data handlers are implemented in both the Android and the UE environment.

#### **In Android**

The two data handlers are implemented as two libraries written in Java and Kotlin. The visual data handler is compiled with the visual process, whereas the network data handler is compiled with the network process. Both data handlers have a transmitter service and a receiver service, for which the transmitter service of the network data handler binds to the receiver service of the visual data handler and vice versa. Data transmitted between the two handlers are shared through Android shared memory (`ashmem`). At app level, developers only need to initialize the two handlers in each process accordingly. Then, they can use the following APIs to send and receive data securely between split processes.

To receive data, we use Android's AIDL feature to create an `onData()` listener for monitoring and receiving the incoming data. Then, we expose a `Receiver<>` interface in the data handler to be registered with the desired string identifier and the data to be received as `ByteArray`.

```
fun onReceived(id:String, data:ByteArray) {}
```

Similarly, we expose a `Send()` interface to transmit the content through the data handler service as `ByteArray` associated with the desired identifier as string.

```
fun send(id:String, data:ByteArray) {}
```

## In UE

Data handler implementation involves: *(i)* exposing UE Blueprint callables, *(ii)* transferring data between the UE-Android boundary and exposing Android Java APIs.

First, the two data handler plugins are implemented in C++ to expose UE Blueprint callable functions for transmitting validated AR library outputs. For example, the following `LensCap` function can be called in UE Blueprint to collect the camera pose acquired from the AR library.

```
UFUNCTION(BlueprintCallable, meta = (DisplayName = "
    LensCap_GoogleARCore_Collect CameraPose", HidePin = "
    LastPose"))
static void VDH_Send_Camera_Pose(FTransform& LastPose);
```

Then, to pass data between the UE-Android boundary, `LensCap` plugins implement send and receive APIs through Android JNI. Currently, we provide support for UE-Android compatible data types, such as `int`, `float`, and `bool`, which are sent and received in the form of arrays. The send JNI function checks if the caller has a valid tag (to differentiate `LensCap ARCore` wrappers), converts UE data to JNI types, and is exposed to Android to connect data to or from UE.

#### 4.6.4 Screen-Based Overlay Composition

We utilize the Android `WindowManager` to overlay the network process screen on the visual process screen. Although `WindowManager` allows the network process to draw its overlay over the visual process screen, the network process cannot observe the pixels of the visual process screen, preserving visual privacy. Adding the screen overlay requires `permission.SYSTEM_ALERT_WINDOW`, subject to user’s approval at runtime. For correct overlay, the layout of the network process screen matches the visual process screen, such as width and height.

Inside the network screen overlay, we are able to implement on-screen touch interfaces to initiate user interactions with the visual screen. To do so, we override the `onTouch(view: View, motionEvent: MotionEvent)` function in the network process to send touch coordinates to the visual process. If developed in UE, the visual process translates the received touch coordinates into the UE coordinate system, which will finally be stacked and processed in UE’s Android input interface. From the developer’s perspective (and user’s perspective), the app with the screen-based overlay operates exactly the same as the legacy application.

#### 4.6.5 Fine-Grained Data Monitoring

The data usage monitor is linked to the Android visual data handler and the UE plugin with the following steps. First, we manually inspect the vision library API to determine the functions to be monitored. In general, the goal of an AR framework is to provide functionalities that operate on camera frames to determine trackables and estimate surroundings. In UE, the `GoogleARCoreFunctionLibrary` only contains hundreds lines of code and 50 functions written in C++, among which we focus on monitoring functions that (i) operate on camera frames, (ii) have a return value,



(iii) work on a block of memory. After narrowing it down, only 25 functions need to be wrapped. Overall, the workload for inspecting the vision library is trivial, even when the vision library needs to be updated for a newer version iteratively. Note that not all of these functions may expose user privacy at a serious level. This presents a future research opportunity to investigate better UI for data usage monitoring that eases the user’s burden to grant permissions at runtime, e.g., grouping functions exposing similar types of data and assign each group a risk level Android Developers (2021a).

Second, we write a wrapper around each function being monitored. The wrapper function is statically tagged. Upon invocation, the `LensCap` user permission is checked. If the user allows specific visual feature collection, the monitored function is then called. Next, this wrapper gets each value from the output, concatenates them, stores them for a data integrity check, and sends them in the format of data array, together with the function tag, through the `Visual→Network` channel provided by the visual data handler.

Third, based on the tag of the function, we add a property into a counter struct `LensCapCounterStruct` and a date struct `LensCapDateStruct` in the visual data handler accordingly to record how often and when the monitored function is called.

Finally, to check the data integrity before `send()` in the visual data handler is called, we directly expose a native function from the UE binary library to our visual data handler, which sends the data to be transmitted back to UE and compares with the original data copy stored. Transmission to the network data handler occurs only if the data match.

The data usage switch notifies users about what visual data is used in the network process and further presents interfaces for users to allow or disallow the transaction of each visual feature. The implementation involves a user permission inquiry

and verification mechanism, as well as a notification interface. Figure 4.3 shows an AR app Google AR (2021) with the data usage switch notifications shown in settings (in the middle). In this example, the data usage switch allows users to disable the Visual→Network transmission of camera pose, lighting estimation, face tracking, point cloud, and camera frame, which are the five use cases evaluated in §4.7.

For permission inquiry and verification, we implement `LensCapPermissionStruct` in the visual data handler to store `boolean` values for each type of output from AR library, as well as a function `boolean getPermTag(String tag)` to respond permission inquires. Then, we expose a permission inquiry function to UE through JNI again to help validate the transmission of a specific visual feature.

LensCap visual data handler service leverages the Android notification channels to present ON/OFF switches for users to selectively allow/disallow the transmission of a monitored visual feature. An ON indicates that the user approves the corresponding visual feature to be transmitted to the network process and the transmission will further be stored in the timer allowing users to analyze in detail about Time-of-Collection in the form of a bar chart. On the contrary, when a notification channel is disabled, values in `LensCapPermissionStruct` will be updated accordingly to prevent visual data from being transmitted.

## 4.7 Evaluation

In this section, we demonstrate the performance of our system in various cloud-based AR apps, comparing legacy single-process app behavior with split-process LensCap app behavior, along with an interview-based user study, to ensure that visual privacy can be preserved without sacrificing app performance or user experience at runtime.

### 4.7.1 Benchmark Applications

To cover popular AR use cases, we build and evaluate five cloud-based AR apps that share different types of visual AR data across multiple devices, based on examples from Google AR (2021). These apps are developed in UE (v4.24) and then deployed to Android devices (Google Pixel 4 XL). A local desktop server serves as a cloud server, storing and passing data among mobile devices. Uploading and downloading data uses an OkHttp client implementation Square, Inc. (2021) with a WiFi connection.

The first app shares the *camera pose*, containing the estimated camera location and rotation, which is critical to tracking and rendering. Collaborative AR apps could share camera poses to achieve shared user viewport geometry, improve camera calibration, or provide runtime user/object tracking Schmalstieg and Hesina (2002). The second app shares the *lighting estimation*, encoding environmental illumination towards rendering virtual objects realistically. Multiple AR users could share radiance samples from multiple perspectives to achieve more accurate lighting estimation for more immersive rendering Prakash *et al.* (2019). The third app shares the AR *point cloud*, which contains the 3D visual corner points that are used to track the space. AR apps can share point clouds among users for shared positioning and/or send it to the cloud for object detection Chen *et al.* (2020) and/or image-based localization Speciale *et al.* (2019). The fourth app shares the *face tracking* result. Face tracking detects and tracks the image regions of faces between camera frames, sharing these over the network, e.g., for identity verification. The last app relies on sharing the *full frame*. In this model, AR apps offload camera frames to the cloud, e.g., for live-streaming, video chat, social media, or cloud-/cloudlet-based vision processing.

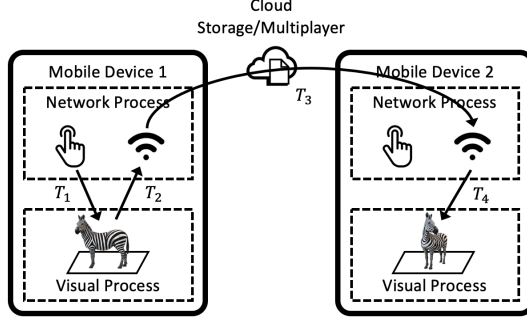


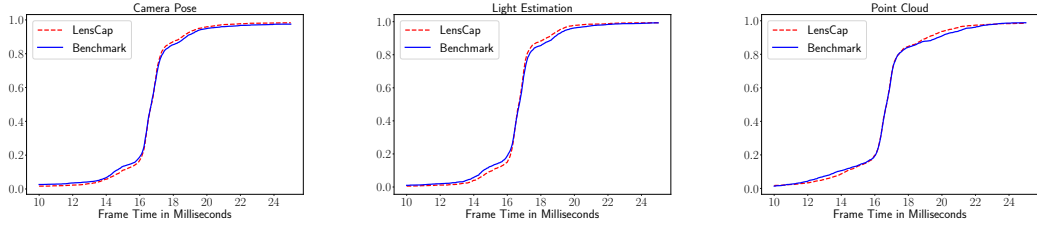
Figure 4.5: We evaluate the interactive latency with a combination of  $T_1$  (Touch→Visual),  $T_2$  (Visual→Network),  $T_3$  (Upload&Download), and  $T_4$  (Network→Visual).

#### 4.7.2 Evaluation Metrics

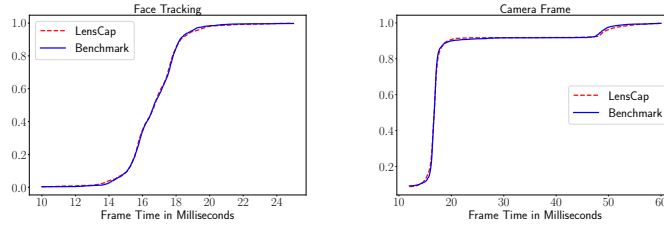
To explore LensCap’s influence on application performance, we monitor the time interval between consecutive frames  $T_f$  (inverse to the frame rate).  $T_f$  is measured by the *deltaTime* (app time elapsed between frames) acquired from the `Tick()` UE Blueprint function, which is called every frame.

In addition, we measure and compare the interactive latency of four time intervals,  $T_1$  to  $T_4$  when running as the legacy single-process benchmark apps and as LensCap-enabled split-process benchmark apps. Depicted in Figure 4.5, the time intervals are defined as follows:

1.  $T_1$  represents the time elapsed between a user behavior and a visual rendering event.
2.  $T_2$  represents the time elapsed to transfer data from the visual process to the network process, which includes several actions, i.e., visual process to visual data handler, visual data handler to network data handler, and network data handler to network process.



(a)  $T_f$  sharing camera poses. (b)  $T_f$  sharing light estimations. (c)  $T_f$  sharing point clouds.



(d)  $T_f$  sharing face regions. (e)  $T_f$  sharing camera frames.

Figure 4.6: CDFs of  $T_f$  for collecting five different types of visual data in every 10 frames demonstrate no performance overhead comparing between the LensCap-integrated apps and the legacy apps.

3.  $T_3$  represents the roundtrip time elapsed to upload and download data between the network process and the cloud component, e.g., sending face detection results to the cloud and receiving a response.
4.  $T_4$  represents the time elapsed between when the network process acquires data from the cloud and when it is applied to the visual process, e.g., utilizing light estimation to improve rendering. ( $T_4$  and  $T_2$  are similar but reversed.)

All time intervals are measured by calculating the difference between system timestamps. We synchronize the system clock between the cloud server, the Android device, and the UE environment. For each app, we run the experiment for 2 minutes

	Camera Pose	Lighting Estimation	Point Cloud	Face Tracking	Camera Frame
	Bench./LensCap	Bench./LensCap	Bench./LensCap	Bench./LensCap	Bench./LensCap
$T_1$	8.9/9.1	8.4/8.7	8.7/8.9	8.7/9.0	8.7/9.0
$T_2$	N/A/0.3	N/A/0.3	N/A/0.3	N/A/0.4	N/A/1.2
$T_3$	99/108	117/121	128/111	124/120	1253/1176
$T_4$	N/A/0.3	N/A/0.4	N/A/0.4	N/A/0.4	N/A/1.3
$T_f$	16.8/16.8	16.7/16.7	16.8/16.8	16.7/16.7	18.4/18.4

Table 4.1: Averaged evaluation results for all time intervals in milliseconds (ms). Note that the interactive latency between processes ( $T_2$  and  $T_4$ ) does not apply to benchmark applications.

to acquire thousands of data samples, during which the data collection is performed roughly every 10 frames, an interval very commonly used in many keyframe-based continuous mobile vision applications Liu *et al.* (2017).

### 4.7.3 Application Performance

We use  $T_f$  to compare and analyze the application performance in each example use case. A comparison of  $T_f$  between the benchmark and the LensCap-integrated app is shown in Figure 4.6 and its averaged value can be found in Table 4.1.

We first evaluate the app performance in the context of programming scenario 3 and 4, in which the network process requests visual data to be uploaded and downloaded from the cloud. In Figure 4.6a, 4.6b, 4.6c, and 4.6d, results show that most  $T_f$  is within 16 ms to 17 ms in both LensCap-integrated and benchmark apps, which indicates a very comparable AR performance that could be maintained at as high as 60 FPS, no matter for collecting camera poses, light estimations, point clouds, or face tracking results. In Figure 4.6e, the result shows that collecting the entire camera frames incurs latency overheads in both benchmark and LensCap-integrated apps. However, the additional latency comes from processing image planes of camera

frames in UE in the visual process. Thus, the overall app performance is the same comparing between the benchmark and the LensCap-integrated apps, i.e., data communication between split processes does not incur noticeable latency overhead. In particular, in this worst case, the app performance could be maintained at around 55 FPS, if camera frames are collected every 10 frames.

In addition, we use app 4 to evaluate the app performance for programming scenario 1, in which the app just detects faces and draws overlays locally in the visual process, without network interactions. Results show that the average of  $T_f$  is 16.8 ms in the legacy app and 16.7 ms in the LensCap-integrated app. Furthermore, we combine app 1, 2, and 3 together into one app to evaluate programming scenario 2, in which the data downloaded from cloud is sent to the visual process repeatedly in a sequence after clicking an on-network-screen button. Results also show similar app performance compared between the legacy app and the LensCap-integrated app (both have an average of 16.7 ms  $T_f$ ).

## Summary

The result demonstrates that the adoption of split-process access control does not appear to influence app performance, likely due to: (i) the computational ability of mobile devices to handle the operation of an additional process, and (ii) the non-blocking data sharing between split processes. Visual privacy can thus be monitored at the process boundary and preserved on the device subject to user's decision, without penalizing the app's performance. From our experiences in the evaluation, the AR experience is robust, smooth, and comparable (without noticing any differences) between LensCap-integrated and benchmark apps.

#### 4.7.4 Interactive Latency

We use  $T_1$ ,  $T_2$ ,  $T_3$ , and  $T_4$  to compare and analyze the interactive latency in each example use case. A comparison of their averaged values across all data samples between the benchmark and the LensCap-integrated app is also shown in Table 4.1.

##### **Camera Pose**

Results show that our system introduces an average of 0.2 ms  $T_1$  Touch→Visual latency for initiating user interactions, as well as 0.3 ms  $T_2$  Visual→Network and 0.3 ms  $T_4$  Network→Visual latency for transmitting tens of bytes of camera pose data between processes.

##### **Lighting Estimation**

Results show that our system introduces an average of 0.3 ms  $T_1$  Touch→Visual latency for initiating user interactions, as well as 0.3 ms  $T_2$  Visual→Network and 0.4 ms  $T_4$  Network→Visual latency for transmitting tens of bytes of lighting estimation data between processes.

##### **Point Cloud**

Results show that our system introduces an average of 0.2 ms  $T_1$  Touch→Visual latency for initiating user interactions, as well as 0.3 ms  $T_2$  Visual→Network and 0.4 ms  $T_4$  Network→Visual latency for transmitting hundreds to thousands of bytes of point cloud data between processes.



## Face Tracking

Results show that our system introduces an average of 0.3 ms  $T_1$  Touch→Visual latency for initiating user interactions, as well as 0.4 ms  $T_2$  Visual→Network and 0.4 ms  $T_4$  Network→Visual latency for transmitting hundreds to thousands bytes of face tracking data between processes.

## Full Frame

Results show that LensCap introduces an average of 0.3 ms  $T_1$  Touch→Visual latency for initiating user interactions, and 1.2 ms  $T_2$  Visual→Network and 1.3 ms  $T_4$  Network→Visual latency for transmitting megabytes of camera data between processes.

## Summary

Split-process access control introduces negligible latency in its inter-process communications, as (i) the 0.2 ms to 0.3 ms overhead in  $T_1$  is all but invisible, compared with the latency needs for gaming and other interactive touch-based applications; according to Jota *et al.* (2013), humans cannot differentiate touch latencies between 1 and 40 ms; and (ii) the 0.3 ms to 1.3 ms latency of  $T_2$  and  $T_4$  caused by the inter-process communication is negligible (without impairing the app performance), even for transmitting the entire camera frame, and even for round-trip operations and interactions across the two processes. On the other hand, cloud communication latency  $T_3$  consumes hundreds of milliseconds (varying based on the network conditions).

### 4.7.5 User Study

Apart from the previous quantitative evaluation, we perform a user study to observe users' hands-on experience of LensCap-integrated AR apps. The user study is

approved by our institution’s IRB. We recruit a total number of 8 undergraduate and graduate students majoring in engineering to participate in this user study. The user study serves three purposes:

- We would like to find out whether the legacy apps and the LensCap-integrated apps have similar performance, from the user’s perspective.
- We want users to freely express privacy concerns while using AR apps and evaluate whether their concerns are mitigated by LensCap.
- We invite users to explore and evaluate the LensCap data usage monitoring UI and brainstorm together with us for a better UI design.

## User Study Procedure

The user study is interview-based, which contains five activity-interview phases, described as below:

1. *Preparation.* In this phase, besides reading the consent form, we asked the participants several questions to get their background in AR. For example, we asked them “What AR applications have you used before?”, “Where do you usually use them?”, and “How often do you use them?”.
2. *Application performance study.* In this phase, we conducted a blind user study, in which the legacy app and the LensCap-integrated app were presented to users in a random order. To make the two versions of apps identical from appearance, we temporarily disabled the LensCap data monitoring module. Participants were given enough time to explore both apps freely, e.g., putting a virtual car model and interact with it. Then, we asked them about their overall experience,

- including “Do you think both apps perform smoothly? If not, which app do you prefer?” and “What differences can you identify between these two apps?”.
3. *Privacy exposure awareness study.* In this phase, we let participants explore the legacy app again. Then, we asked them several questions to understand the baseline of user’s trust in AR apps. These questions included “Imagine that some malicious app developers want to steal your identity, what kind of information in your AR experience do you think they can exploit?”, “What makes you trust or not trust an AR app?”, and “Do you think the current permission model can protect your visual privacy?”.
  4. *LensCap introduction.* In this phase, we educated participants to be familiar with the LensCap app development framework. We explained to participants how LensCap splits the app into two process, how least-privileged split-process access control is managed, how the network screen is overlaid on top of the visual screen, and how users are able to control and visualize the data transactions between process boundaries at a fine granularity. Then, we answered any questions they had.
  5. *LensCap data usage monitoring UI study.* In this phase, we let participants explore the LensCap-integrated app again. We evaluated the current LensCap data monitoring UI from both the usage aspect and the trust improvement aspect. We also invited participants to help us envision a better UI to be design in future that balances the usage and the trust. In particular, we asked participants several questions, including “Do you think LensCap can help you trust untrusted AR apps by allowing you to control what can be collected by the network?”, “With the LensCap functioning logo rendering on the top left, do you feel confident (protected) while using untrusted AR apps even with no

data usage notifications prompted?”, and “What other types of notification or permission models would you like to be deployed that can further improve your trust in AR apps?”.

## User Study Observations

From the interview responses, we garnered the following observations, validating the ability of LensCap from the user’s perspective for maintaining app performance while endowing users with trust in random AR apps:

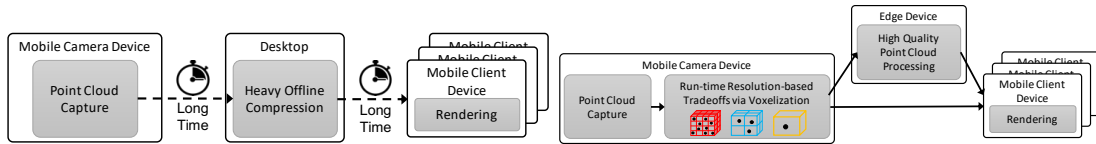
- *All participants are already familiar with AR technology.* They have more or less used AR apps before, in which Pokemon Go and social media apps such as TikTok and Snapchat are the most popular ones.
- *Smartphone is the main AR portal.* Other types of AR devices, such as wearable glasses had been barely used. Users noted that a killer app on those devices would be needed to boost their usage.
- *AR would be used everywhere at anytime.* Though currently AR usage is limited by apps, all participants anticipate to use AR at various places (from home to outside) and some of them even want to use AR all the time, including walking in the street.
- *Participants cannot differentiate between the legacy app and the LensCap-integrated app performance-wise.* All of them agree that the two versions of apps run equally smooth.
- *Participants want AR to be deployed in a wide range of fields.* Apart from simply putting virtual 3D content on top of the real world, participants want

to see AR in “medical settings, to revolutionize surgery”, “education, explaining complicated chemistry concepts by visualizing reactions”, “designing and decorating”, and “gaming and social interactions”.

- *Participants are aware that visual privacy can be exposed in AR apps.* In particular, they are aware that sensitive information such as credit cards, faces, photos left around, and location captured by the camera could be stolen and utilized by malicious AR apps. One participant pointed out that “anything my camera is looking at, can be used for targeted advertisements”.
- *Participants can trust an AR app only if they are given control of the visual stream, e.g., “telling the app not to take my data”, “as long as the visual data stays within the phone”, and “clarification on what gets sent to the network and when”.* All participants think the legacy Android permission model is far from satisfying this condition.
- *All participants felt that LensCap would improve their confidence while using untrusted AR apps.* They already feel safe when LensCap functioning logo is displayed and even safer when notification banners prompted for asking their permissions, with one participant “extremely” liking this setting. Furthermore, all participants think the Time-of-Collection info provided by LensCap is useful.

EDGE-ASSISTED POINT CLOUD LIVE-CAPTURE AND STREAMING  
PLATFORM

Mobile capture of volumetric point clouds enables spatial information in 3 dimensions to be streamed to multiple parties. However, existing solutions for volumetric point cloud capture are too bandwidth-intensive and/or compute-intensive to support real-time streaming operation for dense, high-quality volumetric data. To enable the live-capture and streaming of point clouds on mobile platforms, we propose an edge-assisted point cloud processing framework that can adapt the spatial resolution of targeted portions of the volume to meet performance needs. The core idea revolves around a novel voxel grid schema that represents point clouds as a set of voxel grids, each with a set of parameters that can be tailored to the needs of the users and their applications. The schema enables an adaptive sampling component at the edge device that can sample points flexibly in different voxel sizes and a point cloud resizing component at the client side that can resize and render points captured at different resolutions in the same user view while sustaining visual quality. A comprehensive evaluation demonstrates that our framework maintains a reasonable visual quality with a stable data throughput and a compression ratio comparable to on-device benchmarks, while capable of meeting various performance needs and achieving an improved energy efficiency, ideal for interactive 3D use cases across mobile client devices and mobile networks.



(a) Traditional methods capture point clouds and then compress them offline in two separated stages due to long processing time.

(b) We push light runtime processing to mobile camera devices and offload high quality point cloud processing to edge devices.

Figure 5.1: Our edge-assisted point cloud live-capture and streaming framework utilizes resolution-based tradeoffs on mobile camera devices to enable 3D data visualization on remote mobile client devices in real-time.

## 5.1 Introduction

A point cloud is simply a set of 3D points, capable of representing volumes of captured data. With the rapid development of both depth sensor hardware and visual computing software, point cloud data can be captured at high precision, improving their utility across a variety of applications. Apple recently deployed LiDAR sensors on iPhone, aiming at helping the device to better understand the user’s environment to improve its performance in tasks such as night vision, AR experiences, and machine learning Apple (2021); HUSSAIN (2021); Zhou and Tuzel (2017). Google presented DepthLab Du *et al.* (2020) to ease developers’ burden when using depth data in AR applications on mobile devices. Unity unveiled the Metacast real-time 3D sports platform for creating and delivering interactive content for sports broadcasts Unity (2021). The usage of point clouds portends continuing growth in more immersive applications Thomson (2021); Qi *et al.* (2016); Medina and Paffenroth (2021); Guo *et al.* (2019); Qiu *et al.* (2018); Hackel *et al.* (2017); Xue *et al.* (2021).

However, point clouds are expensive to stream and process, especially compared

to the limited network bandwidth and power of mobile devices. For a typical point cloud format such as PLY Wikipedia (2021), each point consists of three channels position data and three channels color data, usually represented in 12 bytes floating number and 3 bytes byte values accordingly. If the data is captured by a Microsoft Azure Kinect sensor Microsoft Azure (2021) working at 720p and 30 FPS, with the number of points to be processed per frame roughly at 600K, the data throughput can reach 2.2 Gbps, which can hardly be streamed by any types of today’s networks, even 5G. Though some of the most recent mobile devices have equipped with on-board depth sensors such as ToF and LiDAR sensors, the resolution of those sensors is kept low due to the power limit Hu *et al.* (2021b).

To enable the visualization of high resolution point clouds on mobile devices, related works compress point clouds. For example, Draco Google (2021b) and PCL Point Cloud Library (2021) are two tree-based point cloud data processing libraries which convert unorganized raw point clouds into structured tree data to further enable efficient operations such as traversing. GROOT Lee *et al.* (2020) advances the sequential tree structure by proposing a parallel-decodable tree, further improving the computational efficiency. However, as shown in these works, constructing the tree structure from a large amount of point cloud data consumes hundreds of milliseconds to even tens of seconds Hu *et al.* (2021b). Though Kammerl *et al.* (2012) presented a double-buffering tree structure to remove temporal redundancy in point cloud sequences, the task of tracking changes between two trees is still computationally intensive for dense point clouds. Other than the tree structure, Flare Qian *et al.* (2018), Rubiks He *et al.* (2018), and Vivo Han *et al.* (2020) are three frameworks that focus on volumetric data optimizations. However, these works rely on a comprehensive study of existing datasets at an offline stage. They then use the study results to guide the capture of volumetric data in order to reduce the throughput to an amount that is



suitable for mobile devices. These offline study results are hard to deploy in live-capture scenarios, since the prediction accuracy can be low in a live setting and thus degrade the user experience Dasari *et al.* (2020). In Dasari *et al.* (2020), Dasari *et al.* proposed a deep learning-based streaming platform which fetches low resolution 360°-video at the server and reconstructs each frame to super-resolution at the client, working under very limited network bandwidth lower than 30 Mbps.

Comparing with the pre-captured datasets that are used in previous works, point clouds that are live-captured and streamed present common – but also unique – features that we can exploit: *(i)* Across all point cloud datasets, not all points captured are useful. Many works optimize the pre-captured point clouds based on a user’s viewport, such that the hidden points on the back do not have to be streamed. Though visibility-aware optimizations such as viewport adaptation may not be suitable in live-capture, we are still able to separate points into focus and peripheral regions and then treat them with different weights, e.g., environmental points are less important. *(ii)* There is a resolution-based tradeoff to be exploited. Similar to 2D image capture Hu *et al.* (2019), capturing 3D points also presents resolution-based tradeoffs, e.g., capturing less points (subsampling) leads to less processing time but trades visual quality. We can adaptively balance these tradeoffs to maximize the throughput. *(iii)* Point clouds can be represented in different formats to fulfill different requirements, e.g., voxels and trees. In this work, we advocate that the point representation is simply a special case of the voxel representation. We expect to utilize the flexibility provided by the voxel representation to enable fast resolution-based tradeoffs. With this tradeoff distributed across various network components (in particular in a state-of-the-art 5G topology), we can explore a more robust point cloud delivery and rendering system using combined point cloud streaming operations. *(iv)* The user experience is sensitive to the point density and the frame rate. We should include

the user’s preference into the point cloud data processing pipeline, giving users the control of how well they want to visualize the data.

To this end, we propose an edge-assisted multi-resolution adaptive point cloud live-capture and streaming framework tailored for mobile platforms, as shown in Figure 5.1. The core idea behind our framework is to utilize the flexibility of the voxel representation to sample point clouds in different resolutions to exploit the resolution-based tradeoffs among visual quality, energy efficiency, and performance. Then, by fully utilizing edge computing, responsive high-quality visualization can be achieved. As a result, the system empowers users with the capability to visualize high quality point clouds captured off-device with the on-device level of mobility. This enables a variety of interactive apps including AR-based remote coaching, immersive sports viewing, and 3D model reconstruction, etc.

Our framework consists of three major components. First, point clouds are represented in a set of voxel grids and each of them is managed efficiently through a tailored *voxel grid schema*. Each voxel grid is reconfigurable through a set of parameters including the resolution, the pose and bound, and the timing budget. Then, based on the number of voxels, each voxel grid is encoded with dynamic bit depth to save bandwidth while streaming. Second, our framework enables *programmable voxelization*, which adaptively offloads the voxelization operation and the raw data between the mobile camera device and the edge device depending on different network configurations. This enables early processing for interactive activities on the mobile camera device and intensive computations on the cloud for high quality rendering. Finally, on the client side, our framework equips a *point cloud resizing* component to dynamically adjust particle size based on voxel size and then render points represented by different resolutions in a single viewport.

We prototype our framework using three off-the-shelf devices, consisting of an

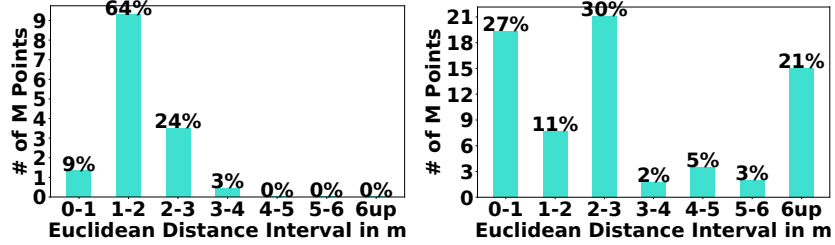
Azure Kinect Microsoft Azure (2021) to capture high resolution/precision point clouds, a Jetson TX2 development board Nvidia developer (2021) to simulate the early data analysis and preparation for utilizing real-time resolution-based tradeoffs on the mobile camera device, and a smartphone to run an interactive AR application. Despite being evaluated with specific hardware, our framework can work on other platforms, including other depth sensors, edge computing platforms, and mobile devices. We evaluate our framework from two aspects. First, we evaluate the effectiveness of our adaptive sampling component in terms of the data throughput, the “compression” ratio, and the energy efficiency. Second, we evaluate the visual quality of our point cloud resizing component in terms of SSIM. The evaluation demonstrates that our edge-assisted framework is able to deliver point clouds at a comparable high throughput meeting various real-time requirements with an improved performance, compression ratio, and energy efficiency, while maintaining a reasonable visual quality.

## 5.2 Motivation

In this section, we conduct a data study offline to *(i)* understand the point cloud data distribution in two datasets, *(ii)* consider differences among the Voxel, the Point, and the Octree representation, and *(iii)* observe the discrepancy in visual quality of rendering point clouds at different resolutions. Though we target live-capture scenarios, studying the data statically reveals common applicable findings.

### 5.2.1 Point Cloud Data Distribution

We study the point cloud data distribution using two datasets. The first one is a publicly available dataset called the RGB-D Scenes V2 Dataset Lai *et al.* (2012) which contains furniture and objects in an indoor environment. We capture the second dataset by ourselves, using an Azure Kinect sensor in an indoor environment with a



(a) Distance distribution of the public RGB-D dataset. (b) Distance distribution of our Kinect dataset.

Figure 5.2: The data distribution of two datasets indicate that most points are within a specific distance range away from the camera center.

human subject performing various tasks, e.g., waving one or both hands, sitting, and walking.

We first study the distance distribution of each point relative to the camera center. The result is shown in Figure 5.2. In the public RGB-D Scenes V2 dataset, as shown in Figure 5.2a, 64% of the points are within 1 to 2 meters from the camera center, which is usually the main object in the center of the view. In our Kinect dataset, as shown in Figure 5.2b, 41% of the points are within 1 to 3 meters from the camera center, which often indicates the area of the human subject is moving within. Both results indicate a strong correlation between the main object and its point’s location.

Then, we study how many points are needed to represent and render high quality point clouds. In an example in the public RGB-D dataset, as shown in Figure 5.3b, if the user is focusing on viewing the chair, the working space for points compositing the chair can be reduced to roughly 30% of the total data. Similarly, in our own dataset, as shown in Figure 5.3a, the main human subject consists of 181381 points (out of 585170 points) which potentially need to be rendered at a high resolution. If we represent the scene using voxels, we can use the same quantity of high-resolution voxels for points compositing the human and 13178 low resolution voxels for peripheral

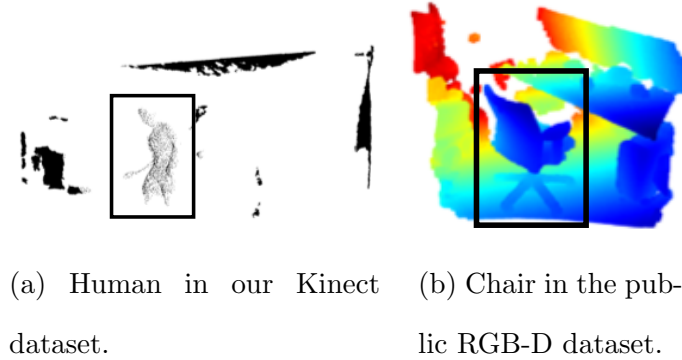


Figure 5.3: Points compositing the human subject and the chair object both occupy roughly 30% of the total point cloud data, which potentially are required to be represented and rendered at a high resolution.

points (59% of the total data), resulting in a compression ratio more than 3.2. We can further increase the compression ratio by dividing the human subject into parts and weighing them differently with different resolutions.

### 5.2.2 Voxel vs. Point vs. Octree

3D point clouds are generated from raw 2D color and depth map. Though color maps in a video sequence can be compressed, streamed, and decompressed at real time in formats such as MJPEG, the compression of depth maps remains inefficient, hardly meeting the real-time specification. In this section, we study the depth (position) data representation among three formats, the Voxel, the Point, and the Octree. We study them using two factors, the memory footprint and the latency it takes to construct. Here, voxelization is performed using open3D Zhou *et al.* (2018) and Octree transformation is performed using Draco Google (2021b).

For memory footprint, the Octree structure is the most efficient data representation. Both Point and Voxel represent point clouds in XYZ (position) and RGB (color) format. The XYZ data in Point usually consists of three 4 bytes `float` to document

the real world position of each point. The XYZ data in Voxel usually consists of three 4 bytes `int` indicating the index of each voxel. Then, both of them add the RGB color data which usually consists of three 1 byte `uint8_t`. For example, in our experiment, point clouds captured at 720p-unbinned can consume as much as 14.7 MB memory if purely stored as Point. Similar memory footprint will be incurred if they are represented by Voxel with small size. However, if point clouds are encoded with Octree, the memory consumption can be one magnitude lower (to 1.3 MB). Representing point clouds by Voxel with large size can also reduce the memory footprint, but points within the same voxel are averaged and their information is lost.

For latency to construct, the Point takes the least amount of time, since it just needs a 2D to 3D transformation. Converting points into Voxel and Octree both take additional time. In our experiment using Open3D Zhou *et al.* (2018), converting a point cloud containing 190K points into Voxel takes 24 ms when voxel size is set to 10 (output 5K voxels), 42 ms when voxel size is set to 5 (output 15K voxels), and more than 2 s when voxel size is set to 1 (output 190K voxels, roughly no optimization). Meanwhile, converting the same amount of points into Octree takes 241 ms, 354 ms, and more than 9 s for maximum depth set to 1, 3, and 8 accordingly. Both operations are CPU-intensive and the proportionality between the amount of points and similar latency can also be identified using the PCL Point Cloud Library (2021) library.

### 5.2.3 Point Cloud Rendering

We explore the resolution implication on rendering point clouds on mobile platforms from two experiments: (i) voxelize all points uniformly with one resolution; (ii) voxelize points separately using a combination of resolutions within the area containing the human subject. Similar to §5.2.2, we capture static point clouds and convert the color and depth images to polygon files. To voxelize the data for downsampling,


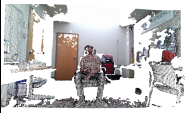
GT	1	16	1/16
			
GT	0.9999	0.6579	0.9187

Figure 5.4: Uniformly voxelizing point clouds with a bigger voxel size (16) will degrade visual quality (in terms of SSIM value). However, the visual quality can be maintained in a multi-resolution setting (1/16), with foreground and background represented separately.

we use Open3D again. To evaluate the visual quality, we use SSIM Horé and Ziou (2010); Hu *et al.* (2020).

First, we downsample the entire point cloud at a voxel size of 4, 8, and 16. As shown in Figure 5.4, the SSIM drops rapidly as we increase the downsample factor, demonstrating quality loss. Second, we separate the point cloud into the foreground and background, with the foreground containing the human subject. The result shown in Figure 5.4 indicates that rendering in a multi-resolution setting can effectively maintain a reasonable objective quality metrics while requiring less input data and consuming less computational resources.

Alternatively, mesh reconstruction can be used to render a cohesive point cloud with minimal holes. However, reconstructing meshes is prohibitively expensive. The ball pivoting algorithm introduced by Bernardini *et al.* (1999) is a popular method used in Open3D. When reconstructing a point cloud with 240K points, the algorithm took 6.28s to render the final image on the CPU, Similar works running on the GPU have similar rendering speeds; Buchart *et al.* (2008) takes 10.27s to reconstruct a 240K point mesh.

### 5.2.4 Findings

From the studies above, we have the following findings.

1. *Resolution-based tradeoffs can be the key enabler to live-capture and streaming.* If point clouds are weighted and divided into tiles represented at different resolutions, e.g., determining the resolution spatially based on the position of the main object (high resolution) and the peripherals (low resolution), the efficiency to process, stream, and further render them can all be improved. Especially for mobile platforms, a sweet spot of the resolution combination to represent the point cloud needs to be determined at runtime to maintain a balance among streaming latency, rendering latency, visual quality, and power efficiency.
2. *The Point representation is simply a special case of the Voxel representation, with a finite range and a high precision.* In this special case, the position of each point is equal to the index of each voxel.
3. *The Voxel representation presents a flexible runtime tradeoff that can be utilized in terms of latency, memory footprint, and rendering quality.* The Point representation does not require any additional latency for processing, however it consumes more memory and takes a long time to be streamed. The Octree representation is less memory intensive, but it incurs unacceptable latency overhead to construct. The Octree structure mainly focuses on easing data traverse for efficient operations such as finding nearest neighbors. It needs to be decoded before rendering. However, the Voxel representation presents a resolution-based tradeoff, i.e., larger voxel size leads to more aggressive optimization with less memory consumption and less computational and streaming latency.

Those findings motivate a voxelization-based point cloud live-capture and stream-



ing framework tailored for mobile platforms that can adaptively define focus regions and select appropriate spatial-temporal resolutions to capture, process, and stream the least amount of points. According to a previous work Hu *et al.* (2021b), point cloud capture and streaming latency, as well as the energy consumption per frame on mobile client devices is proportional to the number of points, e.g., roughly a 70% reduction in latency and energy consumption can be achieved if the system is working on 30% of the total points.

### 5.3 System Design

Motivated by the case study, our system represents point clouds in *voxel grids*, a set of points parameterized by a spatial region workspace and quality resolution (§5.3.1). We design the voxel grid schema to enable efficient processing through resolution-based tradeoffs, such that mobile camera devices can flexibly capture and stream dense point clouds with focused areas of interest for other mobile client devices to receive and render. The streaming applications can set voxel grid parameters, driven by the semantic needs of the use case. Based on the voxel grid parameters, mobile camera devices can sample and voxelize point clouds using multiple resolutions through an adaptive sampling component such that the system can efficiently encode and stream the information (§5.3.2). The receiving mobile client device resizes and renders the points of the voxel grids in a unified viewport (§5.3.3). Figure 5.5 shows an overview of our framework.

#### 5.3.1 Efficient Voxel Grid Schema

Dividing the 3D space into a set of voxel grids enables our framework to independently weight and treat groups of points to utilize resolution-based tradeoffs, managed by an efficient voxel grid schema. The voxel grid operates on a set of parameters,

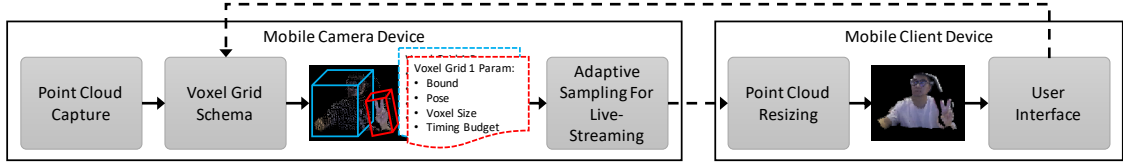


Figure 5.5: Our framework represents 3D point clouds in voxel grids, manages each voxel grid with a set of parameters, and encodes each voxel grid efficiently for live-streaming through a novel voxel grid schema. On mobile client devices, point clouds are resized and rendered in one viewport. Finally, the user interface allows users to customize parameters for each voxel grid.

including the bound and the pose, the resolution, and the timing budget. Then, each voxel grid is encoded separately for efficient live-streaming.

### Bound and Pose

The bound defines the boundary of each voxel grid in the 3D space. Points falling into a particular bound belong to the same voxel grid and thus are processed with the voxel grid’s set of parameters. Specifically, after points are included by a voxel grid, the coordinates are transformed into a local system. This enables voxel grid-based pose (translation and rotation vector) to be applied to all points within. User applications can select a voxel grid and move/rotate it separately from other point clouds.

### Resolution

Resolution indicates the density to represent the volumetric data. In our framework, the context of resolution is inverse to the voxel size. Therefore, a higher resolution means more points need to be sampled by smaller voxels, vice versa. Our framework tracks the requested resolution of each voxel grid and selects a combination of

resolutions to sample points efficiently.

### **Timing Budget**

We monitor the local timing budget for each voxel grid, as well as the global timing budget to meet frame rate expectations. The system can set the global timing budget to meet an overall frame rate. A higher frame rate indicates a lower timing budget, reducing the amount of points that can be sampled in high resolutions. On the other hand, a lower frame rate gives more room for more points to be sampled in high resolutions. At runtime, the system can balance the resolutions of voxel grids such that the combinations of local timing budgets do not exceed the global timing budget.

### **Dynamic Voxel Grid Encoding**

Since a 3D point cloud is represented in voxel grids of varying resolutions, it is important to efficiently represent the combination of voxel grids. Rather than simply using a fixed bit depth for each channel, we introduce a dynamic voxel position encoding solution to quantize the bit depth ( $BD$ ) of each voxel grid as necessary, based on its unique voxel size and bounding box. The idea behind it is simple – for a particular data type, not all bits are fully used for data representation. The system can remove unused bits to reduce memory footprint while maintaining the data completeness. Dynamic voxel position encoding thus decreases the bit depth if fewer voxels are within a voxel grid. For example, 16 bits (in the Point representation) are not needed to index 1000 voxels if they are aligned in one axis; 10 bits suffice. The total number of bits necessary can be described as the sum of the number of bits required to index in each dimension, as listed in Equation 5.1. To decode the location of each voxel, the client simply needs to know the bit depth of each channel and then perform proper

masking.

$$\text{Bit Depth}_{XYZ} = \lceil \log_2 N_X \rceil + \lceil \log_2 N_Y \rceil + \lceil \log_2 N_Z \rceil \quad (5.1)$$

### 5.3.2 Adaptive Sampling for Live-Streaming

The efficient voxel grid schema enables an adaptive point cloud sampling component that is optimized for live-streaming. At runtime, we define voxel grids based on two principles: (i) attempt to only sample points within (or near) the user’s view frustum, and (ii) allow semantic understanding and viewing camera distance to guide resolution and timing budget associated with each voxel grid. Fundamentally, adaptive sampling saves the bandwidth from the source, leading to more efficient data processing, memory operations, and streaming.

#### **Adaptivity to User View**

To monitor the view frustum, the mobile camera device needs to collect the user’s viewing pose and field of view, relative to the point cloud. This forms the view frustum, which determines whether particular points are to be processed or not, i.e., points falling out of the view frustum will be discarded without processing and streaming. Note that there exists an obvious correlation between frame rate and interactive latency because of the frame delivery pipeline (one frame in our case), especially since our system does not predict the user’s viewport movement.

The spatial-resolution can be determined both actively (locally) and passively (globally), from the user’s perspective. Actively (locally), the user can assign each voxel grid with a different resolution by controlling the voxel size. The smallest voxel size indicates the highest resolution, e.g., creating a one to one matching between each voxel and each point. As the voxel size increases, the resolution decreases, and more points are represented by each voxel (with their color information averaged). Passively

(globally), sampling points should obey the overall timing budget, e.g., 33.3 ms for 30 FPS, to maintain an uninterrupted user experience. The timing budget may be large enough for all relevant points to be captured at the highest resolution indicated by the resolution request. However, if it is not, our framework selects a layered spatial-resolution and samples points in multiple resolutions, based on the distance between each point and the user’s camera center. To do so, points far away from the camera center are sampled at a lower resolution with bigger voxel size. After spatial sampling, voxel grids represent point clouds with different bounding sizes, each associated with a unique set of parameters.

### **Adaptivity to Network Topology**

Voxelization is a costly operation that scales with the resolution (voxel size). Performing voxelization at high resolutions completely on mobile camera devices is infeasible due to their low computing power. Offloading expensive operations to networked cloud components may yield high computing power but introducing longer latencies to reach the user. Here, we discuss two policies around the edge and the cloud components accordingly to utilize their capabilities for different use cases.

On the mobile camera device, light voxelization using larger voxels should be performed for *latency-sensitive* use cases. In this scenario, the edge device trades point cloud resolution for performance and interactivity, subject to the timing budget set by the user. This scenario is extremely useful when initial scene understanding is needed such as a quick segmentation for object detection or coarse-grained joint detection, tracking, and related user interactions through storing and retrieving key frames on the edge device itself. Light voxelization on the edge device enables: *(i)* efficient early processing on point clouds to reduce data traffic, especially if there are multiple edges and users in the environment; *(ii)* data processing closer to the user device leading to

less round-trip network latency, which will unleash the potential of various interaction-based use cases including augmented reality; and (iii) an ecosystem around the mobile camera device for fine-grained visual data control.

On the cloud component, heavy/lossless voxelization with smaller voxels should be performed for *quality-sensitive* use cases. In this scenario, the compressed 2D color and depth data can be transmitted from the edge device to the cloud using a lower bandwidth. Then, the cloud component can perform heavy processing on those data such as reconstructing and maintaining the whole point cloud at high resolution in each frame. When needed, the high resolution point cloud can be transmitted to the user’s device in different ways, including encoding voxels into a compressed 2D texture for remote rendering. This scenario can further realize point cloud rendering on the user’s device in a hybrid mode with collaboration between the edge and the cloud. This scenario is useful when the user’s need of visual quality is more important than real-time interactivity, e.g., in an AR-based coaching application when coaches and athletes use a “replay mode” to observe the accuracy of an activity.

### 5.3.3 Point Cloud Resizing

On mobile client devices, rendering points sampled at different resolutions using uniform sized particles will manifest in unbalanced point density in different regions. In a high resolution voxel grid, denser points are sampled and thus the shape is more intact and consistent. On the contrary, in a low resolution voxel grid, fewer points are sampled and rendered which will create visible gaps among them. These gaps will degrade the visual quality and associated user experience if unbalanced point clouds are rendered simultaneously in the user’s view. Thus, instead of receiving and rendering points without the knowledge of points being sampled by different resolution voxel grids, our system dynamically adjusts the point size (resizing particles) at run-

time based on the voxel size associated with each voxel grid schema through a point cloud resizing component. The idea behind this is simple – our system resizes each particle to be large enough to cover the empty space around it to form and present a continuous shape, driven by the associated voxel size.

Unfortunately, resizing particles will not fill in lost pixel information that was averaged into voxels since the system discards raw depth/color sensor data in our current implementation. The system also discards the position and color info of multiple points, lost during the voxelization averaging. Instead of resizing particles with uniformed color, it could be possible to utilize the color texture to interpolate the surrounding pixels (one voxel multiple color) to recover missing information. Compared with 3D position data, 2D color data is very efficient to be streamed in compressed format such as MJPEG. We will explore such functionality in future work.

## 5.4 Implementation

The implementation involves the mobile camera device and the mobile client device. The mobile camera device is implemented in the C++ environment on a Jetson TX2 board installed with Ubuntu v18.04 with Linux kernel v4.9, connected with an Azure Kinect sensor through USB. The mobile camera device implementation involves a multi-threaded data processing pipeline and an adaptive sampling component. On the mobile client device, the implementation, including the point cloud resizing component, is built with C# in the Unity game engine on a Windows machine, which creates an application that we deploy to an Android device.

### 5.4.1 *Multi-Threaded Data Pipeline*

To maximize the throughput, we implemented our framework as a multi-threaded pipeline consisting of four stages, raw sensor data processing, point cloud process-

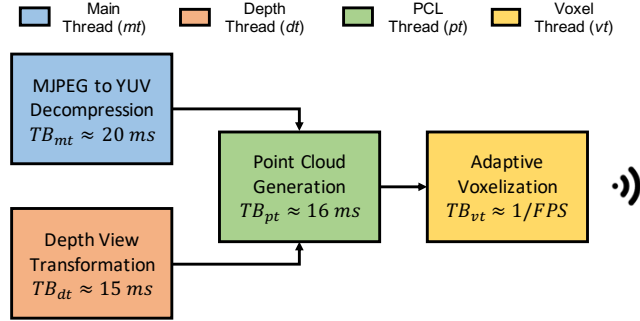


Figure 5.6: Voxelization is pipelined into four stages and the overall throughput of the mobile camera device is determined by the most expensive thread (example latencies show processing at 720p-unbinned).

ing, voxelization, and streaming. Each stage is implemented as its own thread: a color data decompression thread ( $mt$ ) (performed on the main thread), a depth data transformation thread ( $dt$ ), a point cloud generation thread ( $pt$ ), and an adaptive voxelization thread ( $vt$ ), as shown in Figure 5.6. For example, the code below creates a thread object (`vtObj`) for  $vt$  to voxelize point clouds which operates on three pointers (high, medium, and low res point clouds) together with the intended voxel size.

```
std::thread vtObj(generateVoxelWrapper, pclH, pclM, pclL,
                 voxelSizeH, voxelSizeM, voxelSizeL);
```

Each stage contains two sets of buffers for exchange. The raw depth and color data are retrieved from the Kinect sensor and then transformed into the same color view through the K4A SDK. To decompress MJPEG image to YUV format, we used `NvJpegDecoder` Nvidia (2021). To send point cloud data, we used a simple `TCP Socket`.

At stage one, the  $mt$  thread takes the color data to decompress MJPEG to YUV and saves the result into  $Color_{buf1}$ ; the depth data is passed to create  $dt$  thread to



transform XY depth into XYZ point cloud and the result is stored in  $Depth_{buf1}$ . At stage two,  $Color_{buf2}$  and  $Depth_{buf2}$  are filled by  $mt$  and  $dt$ , while  $Color_{buf1}$  and  $Depth_{buf1}$  are passed to the  $pt$  thread to generate XYZ-YUV point cloud data which is then stored in  $PCL_{buf1}$ . At stage three,  $Color_{buf1}$ ,  $Depth_{buf1}$ , and  $PCL_{buf2}$  are filled, while  $PCL_{buf1}$  is passed to  $vt$  thread to voxelize point clouds in multiple resolutions and store the result in  $Voxel_{buf1}$ . At stage four,  $Color_{buf2}$ ,  $Depth_{buf2}$ ,  $PCL_{buf1}$ , and  $Voxel_{buf2}$  are filled, while  $mt$  uses the socket `Send()` to transmit  $Voxel_{buf1}$  to mobile client devices. The pipeline of threads enables the framework to send the voxelized data to mobile client devices per frame.

Each thread takes some time to complete tasks, defined as  $T_{mt}$ ,  $T_{dt}$ ,  $T_{pt}$ , and  $T_{vt}$  accordingly. As these are fully pipelined, the maximum throughput is determined by the most expensive thread, which in turn produces the frame rate experienced by the user, as shown in Equation 5.2. Depending on the voxel size, the voxelization thread usually incurs a wide range of latency. Thus, inside  $vt$ , we implemented two sub-threads such that high, medium, and low resolution voxelization can all be processed in parallel and their timing cost can be further balanced and subdivided.

$$\max(TB_{mt}, TB_{dt}, TB_{pt}, TB_{vt}) \leq 1/FPS \quad (5.2)$$

#### 5.4.2 Adaptive Point Cloud Sampling

### The User Interface

As described in §5.3, our framework allows users to interact with each individual voxel grid to define a set of desired parameters. This implementation involves four settings according to our system design, including the interactive pose, the designated bound, the performance target  $FPS_{min}$  (inverse to timing budget), and the resolution

request. For the pose and the bound, we rely on existing interactive functionalities integrated in the game development engine to let users create bounding boxes through gestures and touches and further interact with those boxes through rotating or re-positioning. For the performance target and the resolution request, we expose them as two sliding bars, indicating the scale between the maximum and the minimum. The bar of performance target has a scale from 20 to 100, indicating the minimum acceptable frame per second. The resolution request bar has a scale from 1 to 10, indicating the maximum and minimum resolution, i.e., voxel size. Note again that a lower voxel size equals to a higher resolution.

### **The Spatial Resolution**

The spatial resolution is then adaptively determined for live-streaming, according to the user’s resolution request,  $FPS_{min}$  target, number of points in user-initiated voxel grids, as well as the current camera center, which are all read from the mobile client device and updated per frame. As the top priority, the  $FPS_{min}$  target needs to be met by maintaining a stable throughput. Then, our system samples points within the user-initiated voxel grids using the requested resolution. During sampling, the latency cost ( $C_v$ ) of voxelizing each point in different voxel grid resolutions is documented and updated per frame, in the unit of nanoseconds per point. The maximum number of points can be sampled per frame at each resolution is then determined by  $1/FPS_{min}/C_v$ . If the number of points to be sampled exceeds the throughput limit, our system samples points closer to the camera using the requested resolution, then selects lower resolutions to sample farther points. However, if the system can handle more points in a higher resolution, more points closer to the camera will be sampled at the requested resolution. At last, if the user-initiated voxel grids are made larger or smaller, our system re-evaluates the condition above and then

re-sample points.

## Voxel Grid Encoder and Decoder

In our implementation, the resolution request directly translates to voxel size, e.g., when resolution request is set to 1 (full), it translates to a voxel size equaling to 1, i.e., 1 voxel to 1 point (the same millimeter precision with the raw data from the Kinect). To voxelize point clouds, we utilize the `VoxelGrid` API provided by the Open3D Zhou *et al.* (2018) library, a popular library for 3D processing. To encode each voxel grid for efficient positioning on the server, we use the maximum bound ( $max_{bound}$ ) and the minimum bound ( $min_{bound}$ ) of a given voxel grid and then use the differences between  $max_{bound}$  and  $min_{bound}$  to determine the minimum number of bits that is required to number all voxels in the X ( $BD_X$ ), Y ( $BD_Y$ ), and Z ( $BD_Z$ ) axis. We then shift the voxel ID in each channel to the left according to  $BD_X$ ,  $BD_Y$ , and  $BD_Z$  to fit the voxel location into one variable using the bit-wise OR operator, with its total number of bits rounded up to multiples of 8 for byte-alignment. After the encoding, we attach the voxel size, the data length, the offset and bit depth in each channel, and the bounds to the buffer containing the voxelized data as the header to be used for decoding. In our implementation, all of the header information results in a negligible overhead of less than 100 bytes. On the client, to decode the voxelized data, we simply need to parse the header and utilize bit masks to reconstruct the voxel locations by manipulating bits. For example, to decode the voxel ID for a 32 bits input which has 8 bits  $BD_X$ , 8 bits  $BD_Y$ , and 13 bits  $BD_Z$ , we first create three bit masks for each XYZ channel, e.g.,  $maskX = 1 \ll BD_X - 1$  for voxel location in the X channel. Then, the voxel location in the X channel is reconstructed with right-shift and bit-wise AND, i.e.,  $input \gg (32 - BD_X) \& maskX$ .

### 5.4.3 Point Cloud Resizing

The implementation of point cloud resizing involves shader programming and determining the appropriate particle resizing factor for each voxel size. Based on the header (described in §5.4.2), the client receives and decodes the data buffer, documents the starting indices for high, medium, and low voxel sizes, and stores the point cloud in a custom structure consisting of two `List<Vector3>`; one for color data and one for vertex data.

Then, we feed the color and vertex data to our custom shader through global `ComputeBuffers`. We use Unity’s built-in particle system implementation to render points, which limits the number of particles rendered per system to 16384. Therefore, we instantiate the required number of particle systems to render the full number of points. After that, the starting indices are sent to the shader, as well as the point indices for that specific particle system. Our custom shader runs on each particle system, only rendering the assigned points. The global `ComputeBuffers` are shared among all the shaders, as each one only accesses the data at its assigned point indices. The particle is rendered to the scene as an unlit billboard. Unity’s forward rendering pipeline then renders the scene to the camera.

To determine the optimal particle size for each voxel size, we implement a data-driven approach to study their relationship. We capture point clouds and voxelize them with 10 different voxel sizes, according to §5.4.2. For each point cloud sampled at each voxel size from 2 to 10, we render it with different particle sizes (with a precision of 0.01). Then, we compute its SSIM against the ground truth point cloud voxelized at the size of 1. SSIM is used as a quality metric since it computes the direct pixel difference for the final render. We find that SSIM trends upwards until the particles were big enough to cover the entire point cloud without showing any

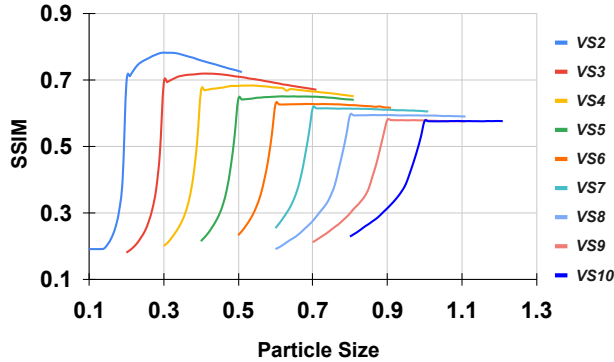


Figure 5.7: For each voxel size (VS), SSIM goes upwards until the particles are big enough and then goes downwards as particles continuously become bigger, i.e., an optimal resizing factor can be identified.

of the background, at which point they began trending downwards, as shown in Figure 5.7. We use the particle size with the highest SSIM value for our subsequent renders. Note that the distance between particles and the camera center has little effect on this relationship, due to the forward renderer accounting for distance when rendering the particle systems in the final image.

## 5.5 Evaluation

We evaluate our framework from two aspects: *(i)* the performance of adaptive sampling and *(ii)* the visual quality of point cloud resizing.

### Hardware Platform

To sample point clouds, we used an off-the-shelf depth sensor – an Azure Kinect Microsoft Azure (2021). The color data is captured at 720p and 1080p, meanwhile the depth data is captured at binned (512x512) and unbinned (1024x1024) at Wide Field-of-View (WFOV). The resolution of the point cloud data is then denoted as color-depth (e.g., 720p-binned) in the rest of the evaluation. To adaptively sample

and process point clouds at the mobile camera device, we used an Nvidia Jetson TX2 development board introduced in §5.4. To stream point clouds, we connected the Jetson TX2 to a TP-Link router tp-link (2021) with a wire. The TP-Link router is further connected to the mobile client device wirelessly, with a download bandwidth as high as 450 Mbps. To resize and render point clouds, we used a Google Pixel 5 smartphone to run a Unity application in the Android environment.

### 5.5.1 The Performance of Adaptive Sampling

#### Dataset

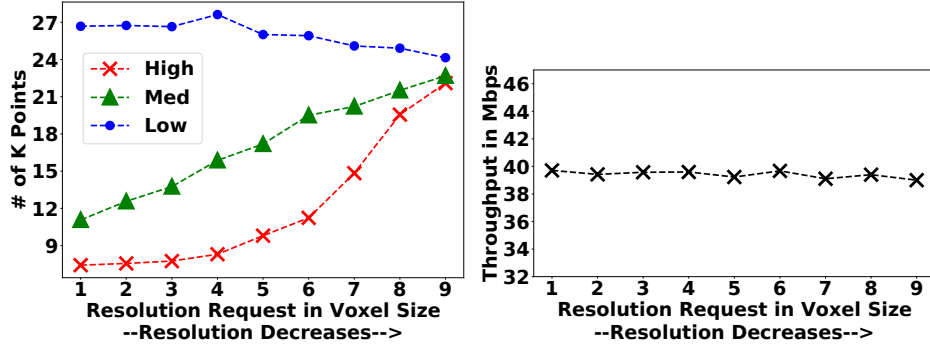
This evaluation is performed in a live-capture and streaming setting in an indoor environment. We simply ask a human subject to perform daily tasks in front of the Azure Kinect sensor, e.g., typing, writing, waving hands, walking, standing up and sitting down, etc. The number of points captured per frame is roughly 600K at 720p and more than 1.3M at 1080p. Each point originally consists of 12 bytes `float` position info and 3 bytes `uint8_t` color info, the same as the popular polygon file format Wikipedia (2021).

#### Metrics

We use three metrics to demonstrate the performance of the adaptive sampling component in our framework, including *(i)* the data throughput, *(ii)* the compression ratio, and *(iii)* the runtime energy efficiency.

#### Data Throughput

For this metric, we perform two experiments to understand *(i)* on the mobile camera device, if our framework can adaptively determine the voxel size to maintain the throughput; *(ii)* on the mobile client device, whether our framework can render with



(a) # of points to resolution.

(b) Throughput to resolution.

Figure 5.8: On the mobile camera device, our system adjusts the voxel size (high and medium) to adaptively sample points according to the resolution request. Voxel grids with lower resolutions cost less bits to encode all voxels, and thus more voxels can be transmitted under the target throughput stably at 30 FPS.

a high data throughput comparable to state-of-the-art solutions. The data throughput is measured in the unit of mega bits per second (Mbps).

The result of this experiment is shown in Figure 5.8, which is averaged across multiple runs each consisting of a 1000-frame period. On the Kinect plus Jetson TX2 mobile camera device, we set the frame rate target to be stable at 30 FPS. Then, according to the resolution request (high res set by the user), our system adaptively determines the number of points to be sampled at a low voxel size and selects the medium voxel size to sample points around the user-initiated voxel grid in a medium resolution, while maintaining the environmental points to be represented stably in a high voxel size at a low resolution, as shown in Figure 5.8a. Because the voxel size increases as the resolution decreases, in the same region, less bit depth is required to number all voxels. As a result, more voxels can be transmitted under the same throughput target. Finally, the overall throughput can be maintained, as shown in Figure 5.8b. On the Pixel 5 mobile client device, we find that receiving, resizing, and

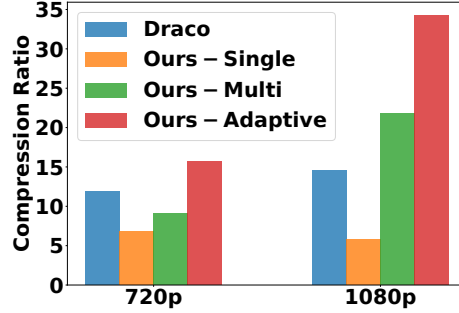


Figure 5.9: Compression ratio of Draco and our framework, with our framework sampling points in one resolution (-Single), multi-resolution (-Multi), and adaptive multi-resolution (-Adaptive).

rendering point clouds can incur a wide range of stable throughput up to 73 Mbps depending on the voxel size and target frame rate. These results are comparable to the wide range shown in Vivo Han *et al.* (2020), although ours have a lower maximum throughput. However, our framework provides runtime resolution tradeoffs which can meet the requirement for a variety of networking conditions. Note that there is a throughput discrepancy between mobile camera device and mobile client device (higher at receiver than at transmitter) which further opens opportunities for a multi-edge environment.

### Compression Ratio

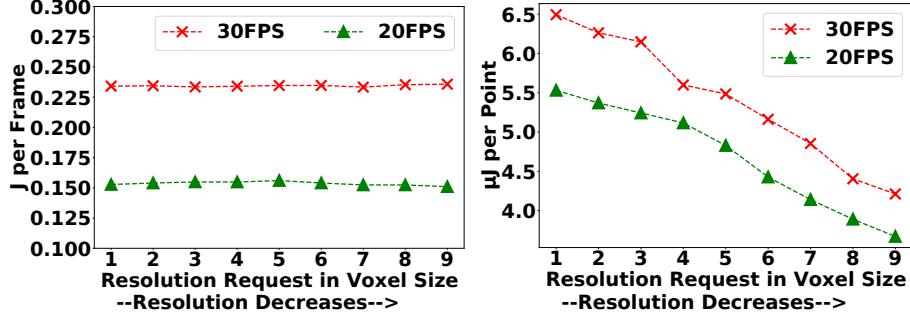
Here, we compare our adaptive sampling framework against Draco Google (2021b) since Draco is demonstrated to have the best compression ratio Han *et al.* (2020). In this experiment, Draco is statically parameterized with the compression level set at 7 and the quantization bit set at 11. On the contrary, our framework is running in a dynamic setting, in which the resolution is randomly set every 50 frames. Note that our framework does not “compress” point clouds but voxelizes and samples point clouds using different voxel sizes.



Running with our framework, we first include all points from the capture and represent the whole point cloud using one voxel grid with one voxel size. In this case, the average memory footprint is roughly at 2.3 MB. Then, we represent the point cloud using three voxel grids, each associated with a different voxel size, however, with an arbitrary number of points set to sample high res (100K), medium res (200K), and low res (the rest) points. In this case, the average memory footprint is roughly at 1.7 MB. Finally, we apply adaptivity to each voxel grid such that the number of points to be sampled at different resolutions is dynamically determined. In this case, the average memory footprint is reduced to 0.99 MB. A comparison of the compression ratio is shown in Figure 5.9. The result demonstrates that our adaptive sampling framework can achieve a comparable or even higher compression ratio than Draco, by sampling point clouds using a set of resolutions. In addition to those three cases, we apply focus regions to the scene, in which only points within those regions are sampled and streamed. In this case, the memory footprint can be as low as hundreds KB, which leads to a one magnitude higher compression ratio compared with Draco.

### **Runtime Energy Efficiency**

We evaluate the runtime energy efficiency in terms of the *energy consumption per frame* in the unit of joule (the product of power and frame time) and the *energy consumption per point* in the unit of  $\mu\text{J}$  (the quotient of power divided by the number of points sampled per second) on both the mobile camera device and the mobile client device. The evaluation is performed in the same adaptive multi-resolution setting running with different resolution requests as described in the throughput evaluation, with a frame rate set stable at 30 FPS and 20 FPS. On the Kinect plus Jetson TX2 mobile camera device, we retrieve the power consumption of VDD\_IN by reading its power rail sys file. The power consumption of VDD\_IN includes all major components



(a) J per frame.

(b)  $\mu\text{J}$  per point.

Figure 5.10: A stable data throughput incurs a stable energy consumption/frame and a reduction in energy consumption/point when more voxels can be transmitted with larger voxel sizes. A performance-based energy tradeoff can also be identified.

on board including GPU, SoC, CPU, and DDR. On the Pixel 5 client, we measure the power consumption by reading the voltage (`voltage_now`) and current (`current_now`) sys files of the battery class using remote `adb`. Figure 5.10 shows the result measured on the mobile camera device. We find that (i) a stable data throughput incurs a stable energy consumption per frame, demonstrating that our framework does not incur any power overhead, as shown in Figure 5.10a; (ii) more points can be sampled and transmitted as voxel size increases, presenting a reduction in energy consumption per point, as shown in Figure 5.10b; (iii) there is a performance-based energy tradeoff, i.e., lower frame rate incurs lower energy consumption per frame and per point. Similar trend in energy consumption per frame and per point can be identified on the Pixel 5, with one magnitude lower values.

## 5.5.2 *The Visual Quality of Point Cloud Resizing*

### **Dataset**

In this evaluation, we capture the dataset and perform experiments on it in an offline setting. The dataset contains point clouds captured and rendered in three different settings, including single resolution, multi-resolution without point cloud resizing, and multi-resolution with point cloud resizing. Similar to §5.5.1, single resolution point clouds are voxelized with a voxel size from 1 to 9 and multi-resolution point clouds are voxelized using a layered resolution. To resize point clouds, we use the particle size determined by our study introduced in §5.4.3.

### **Metrics**

To measure the visual quality of the final renders, we use SSIM again to compute pixel differences between rendered images. In particular, we perform comparisons on multi-resolution renders with and without point cloud resizing, against ground truth point clouds sampled using a single resolution.

### **Visual Quality**

When point clouds are rendered without resizing, the average SSIM across all voxel sizes is only at 0.67. In comparison, after point cloud resizing is applied, the average SSIM is improved to 0.78, equaling to a 16.4% improvement in visual quality. We acknowledge that an optimized visual quality metric needs to be developed to take users' different resolution needs into the consideration. We will actively explore it and perform a more comprehensive visual quality study in our future work.

### CONCLUSION

In this work, we presented three solutions towards fine-grained control of visual data in mobile systems for continuous mobile vision applications to operate with energy efficiency improved, visual privacy protected, and 3D data interactivity enhanced. First, for resolution-based energy and accuracy tradeoff, we observe a substantial image sensor resolution reconfiguration latency caused by the sequential reconfiguration procedure in current operating systems. This long reconfiguration latency gives vision applications a perception of losing frames which impedes the adoption of otherwise beneficial resolution-energy tradeoff mechanisms. In this paper, we propose Banner as a system solution for providing rapid and seamless image sensor resolution reconfiguration. Evaluated in three different OpenCV workloads including display-only, cloud-based offloading, and marker-based pose estimation running on a Jetson TX2 board, Banner is able to halve the end-to-end reconfiguration latency and completely remove the frame-to-frame latency, i.e., no frame drop during sensor resolution reconfiguration even for workloads working at 30 FPS. This allows a more than 49% system power consumption reduction comparing reconfiguring the sensor resolution from 1080p to 480p with downsampling 1080p $\downarrow$ 480p. Banner unlocks a variety of mobile vision tasks to dynamically reconfigure sensor resolutions to adapt to the environmental change and then maximize the energy efficiency. Second, for preserving visual privacy, we introduce LensCap, a split-process app development framework to protect user’s visual privacy in cloud-based AR apps. LensCap isolates the processing of camera frames into a distinct visual process, meanwhile maintaining the cloud communication through another network process, with the data transactions between

split processes monitored and shown to users for approval at a fine granularity. We prototype LensCap as an Android library that could be integrated into the AR development flow of Unreal Engine as a plugin. We evaluate LensCap in five UE projects developed for Android platforms. Results collected from the performance evaluation together with an interview-based user study demonstrate that visual privacy could be preserved and user confidence could be improved with LensCap split-process access control implemented in untrusted AR apps, without any noticeable performance penalty. Third, for an enhance interactivity through 3D data, we presented an edge-assisted point cloud live-capture and streaming framework to enable real-time high precision point clouds visualization on mobile client devices. Our framework is built around an efficient voxel grid schema that enables the utilization of resolution-based tradeoffs on point clouds at runtime. Then, based on a set of voxel grid parameters, our framework adaptively sample points in multiple resolutions efficiently for live-streaming. On the mobile client device, points represented by different voxel sizes are resized and rendered in the same viewport. Our prototype using three off-the-shelf components demonstrates that point clouds can be delivered at a stable throughput adaptive to various requirements with the visual quality maintained while working with an improved energy efficiency. With massive visual data controlled by users at a fine granularity, a bloom of new reality through pervasive computing can be expected.

## REFERENCES

- Facebook, “Mixed Reality Capture”, <https://developer.oculus.com/documentation/native/pc/dg-mrc/> (2021).
- Aditya, P., R. Sen, P. Druschel, S. Joon Oh, R. Benenson, M. Fritz, B. Schiele, B. Bhattacharjee and T. T. Wu, “I-pic: A platform for privacy-compliant image capture”, in “Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services”, MobiSys ’16, pp. 235–248 (ACM, New York, NY, USA, 2016), URL <http://doi.acm.org/10.1145/2906388.2906412>.
- AlDuaij, N., A. Van’t Hof and J. Nieh, “Heterogeneous multi-mobile computing”, in “Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services”, MobiSys ’19, p. 494–507 (Association for Computing Machinery, New York, NY, USA, 2019), URL <https://doi.org/10.1145/3307334.3326096>.
- Android Developer, “Media Framework Hardening”, <https://source.android.com/devices/media/framework-hardening> (2021).
- Android Developers, “Permissions overview”, <https://developer.android.com/guide/topics/permissions/overview> (2021a).
- Android Developers, “Secure an Android Device”, <https://source.android.com/security> (2021b).
- Apple, “Use hdr on your iphone, ipad, and ipod touch”, URL <https://support.apple.com/en-us/HT207470> (2018).
- Apple, “Apple unveils new ipad pro with breakthrough lidar scanner and brings trackpad support to ipados”, <https://www.apple.com/newsroom/2020/03/apple-unveils-new-ipad-pro-with-lidar-scanner-and-trackpad-support-in-ipados/> (2021).
- Apple, “Arkit”, URL <https://developer.apple.com/arkit//> (2021).
- Arzt, S., S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Oceau and P. McDaniel, “Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps”, in “Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation”, PLDI ’14, p. 259–269 (Association for Computing Machinery, New York, NY, USA, 2014), URL <https://doi.org/10.1145/2594291.2594299>.
- Ashe, G., “What Mary Meeker’s Internet Trends Report Means for the State of In-Store”, <https://blog.thirdchannel.com/mind-the-store> (2017).
- Backes, M., S. Bugiel, C. Hammer, O. Schranz and P. Von Styp-Rekowsky, “Boxify: Full-fledged app sandboxing for stock android”, in “Proceedings of the 24th USENIX Conference on Security Symposium”, SEC’15, p. 691–706 (USENIX Association, USA, 2015).

- Ben Abdesslem, F., A. Phillips and T. Henderson, “Less is more: Energy-efficient mobile sensing with senseless”, in “Proceedings of the 1st ACM Workshop on Networking, Systems, and Applications for Mobile Handhelds”, MobiHeld ’09 (ACM, 2009), URL <http://doi.acm.org/10.1145/1592606.1592621>.
- Ben Ali, A. J., Z. S. Hashemifar and K. Dantu, “Edge-slam: Edge-assisted visual simultaneous localization and mapping”, in “Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services”, MobiSys ’20, p. 325–337 (Association for Computing Machinery, New York, NY, USA, 2020), URL <https://doi.org/10.1145/3386901.3389033>.
- Berenbaum, S., “Google glass explorer edition has a 30-minute battery life while shooting video”, URL <https://www.digitaltrends.com/mobile/google-glass-30-minute-videobattery/> (2013).
- Bernardini, F., J. Mittleman, H. Rushmeier, C. Silva and G. Taubin, “The ball-pivoting algorithm for surface reconstruction”, *IEEE transactions on visualization and computer graphics* **5**, 4, 349–359 (1999).
- Buchart, C., D. Borro and A. Amundarain, “Gpu local triangulation: an interpolating surface reconstruction algorithm”, *Computer Graphics Forum* **27**, 3, 807–814, URL <https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2008.01211.x> (2008).
- Buckler, M., P. Bedoukian, S. Jayasuriya and A. Sampson, “Eva2: Exploiting temporal redundancy in live computer vision”, in “Proceedings of the 45th Annual International Symposium on Computer Architecture”, ISCA ’18, pp. 533–546 (IEEE Press, Piscataway, NJ, USA, 2018), URL <https://doi.org/10.1109/ISCA.2018.00051>.
- Buckler, M., S. Jayasuriya and A. Sampson, “Reconfiguring the imaging pipeline for computer vision”, in “The IEEE International Conference on Computer Vision (ICCV)”, (2017).
- Chen, J., B. Lei, Q. Song, H. Ying, D. Z. Chen and J. Wu, “A hierarchical graph network for 3d object detection on point clouds”, in “Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)”, (2020).
- Chen, T., H. Balakrishnan, L. Ravindranath and P. Bahl, “Glimpse: Continuous, real-time object recognition on mobile devices”, *GetMobile: Mobile Computing and Communications* **20**, 26–29 (2016).
- Chu, D., N. D. Lane, T. T.-T. Lai, C. Pang, X. Meng, Q. Guo, F. Li and F. Zhao, “Balancing energy, latency and accuracy for mobile sensor data classification”, in “Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems”, SenSys ’11, p. 54–67 (Association for Computing Machinery, New York, NY, USA, 2011), URL <https://doi.org/10.1145/2070942.2070949>.
- Costan, V. and S. Devadas, “Intel sgx explained”, *IACR Cryptol. ePrint Arch.* **2016**, 86 (2016).

- Dasari, M., A. Bhattacharya, S. Vargas, P. Sahu, A. Balasubramanian and S. R. Das, “Streaming 360-degree videos using super-resolution”, in “IEEE INFOCOM 2020 - IEEE Conference on Computer Communications”, pp. 1977–1986 (2020).
- Dawoud, A. and S. Bugiel, “Droidcap: Os support for capability-based permissions in android”, in “NDSS Symposium 2019”, (2019), URL <https://publications.cispa.saarland/2818/>.
- Du, R., E. Turner, M. Dzitsiuk, L. Prasso, I. Duarte, J. Dourgarian, J. Afonso, J. Pascoal, J. Gladstone, N. Cruces, S. Izadi, A. Kowdle, K. Tsotsos and D. Kim, “Depthlab: Real-time 3d interaction with depth maps for mobile augmented reality”, in “Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology”, UIST ’20, p. 829–843 (Association for Computing Machinery, New York, NY, USA, 2020), URL <https://doi.org/10.1145/3379337.3415881>.
- Enck, W., P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel and A. N. Sheth, “Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones”, in “Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation”, OSDI’10, pp. 393–407 (USENIX Association, Berkeley, CA, USA, 2010), URL <http://dl.acm.org/citation.cfm?id=1924943.1924971>.
- Epic Games, “Unreal Engine”, <https://www.unrealengine.com/en-US/> (2021).
- FACEBOOK, “Introducing Project Aria”, <https://about.fb.com/realitylabs/projectaria/> (2021).
- Feng, Y., S. Liu and Y. Zhu, “Real-time spatio-temporal lidar point cloud compression”, (2020).
- Fernandes, E., J. Paupore, A. Rahmati, D. Simionato, M. Conti and A. Prakash, “Flowfence: Practical data protection for emerging iot application frameworks”, in “25th USENIX Security Symposium (USENIX Security 16)”, pp. 531–548 (USENIX Association, Austin, TX, 2016), URL <https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/fernandes>.
- Google, “Arcore”, URL <https://developers.google.com/ar//> (2021a).
- Google, “Draco”, <https://github.com/google/draco> (2021b).
- Google AR, “Google ARCore SDK for Unreal”, <https://github.com/google-ar/arcore-unreal-sdk> (2021).
- Guo, B., “iOS Security”, [https://www.cse.wustl.edu/~jain/cse571-14/ftp/ios\\_security/index.html](https://www.cse.wustl.edu/~jain/cse571-14/ftp/ios_security/index.html) (2014).
- Guo, Y., H. Wang, Q. Hu, H. Liu, L. Liu and M. Bennamoun, “Deep learning for 3d point clouds: A survey”, CoRR **abs/1912.12033**, URL <http://arxiv.org/abs/1912.12033> (2019).



- Ha, K., 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”, MobiSys ’14 (ACM, 2014).
- Hackel, T., N. Savinov, L. Ladicky, J. D. Wegner, K. Schindler and M. Pollefeys, “Semantic3d.net: A new large-scale point cloud classification benchmark”, CoRR **abs/1704.03847**, URL <http://arxiv.org/abs/1704.03847> (2017).
- Han, B., Y. Liu and F. Qian, “Vivo: Visibility-aware mobile volumetric video streaming”, in “Proceedings of the 26th Annual International Conference on Mobile Computing and Networking”, MobiCom ’20 (Association for Computing Machinery, New York, NY, USA, 2020), URL <https://doi.org/10.1145/3372224.3380888>.
- Haris, M., G. Shakhnarovich and N. Ukita, “Task-driven super resolution: Object detection in low-resolution images”, (2018a).
- Haris, M., G. Shakhnarovich and N. Ukita, “Task-driven super resolution: Object detection in low-resolution images”, CoRR **abs/1803.11316**, URL <http://arxiv.org/abs/1803.11316> (2018b).
- He, J., M. A. Qureshi, L. Qiu, J. Li, F. Li and L. Han, “Rubiks: Practical 360-degree streaming for smartphones”, in “Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services”, MobiSys ’18, p. 482–494 (Association for Computing Machinery, New York, NY, USA, 2018), URL <https://doi.org/10.1145/3210240.3210323>.
- He, Y., L. Ma, Z. Jiang, Y. Tang and G. Xing, “Vi-eye: Semantic-based 3d point cloud registration for infrastructure-assisted autonomous driving”, in “Proceedings of the 27th Annual International Conference on Mobile Computing and Networking”, MobiCom ’21, p. 573–586 (Association for Computing Machinery, New York, NY, USA, 2021), URL <https://doi.org/10.1145/3447993.3483276>.
- Hegarty, J., J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz and P. Hanrahan, “Darkroom: Compiling high-level image processing code into hardware pipelines”, ACM Trans. Graph. **33**, 4, 144:1–144:11, URL <http://doi.acm.org/10.1145/2601097.2601174> (2014).
- Hegarty, J., R. Daly, Z. DeVito, J. Ragan-Kelley, M. Horowitz and P. Hanrahan, “Rigel: Flexible multi-rate image processing hardware”, ACM Trans. Graph. **35**, 4, 85:1–85:11, URL <http://doi.acm.org/10.1145/2897824.2925892> (2016).
- Herbster, R., S. DellaTorre, P. Druschel and B. Bhattacharjee, “Privacy capsules: Preventing information leaks by mobile apps”, in “Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services”, MobiSys ’16, pp. 399–411 (ACM, New York, NY, USA, 2016), URL <http://doi.acm.org/10.1145/2906388.2906409>.
- Horé, A. and D. Ziou, “Image quality metrics: Psnr vs. ssim”, in “2010 20th International Conference on Pattern Recognition”, pp. 2366–2369 (2010).

- Hu, J., G. Choe, Z. Nadir, O. Nabil, S.-J. Lee, H. Sheikh, Y. Yoo and M. Polley, “Sensor-realistic synthetic data engine for multi-frame high dynamic range photography”, in “Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops”, (2020).
- Hu, J., A. Iosifescu and R. LiKamWa, “Lenscap: Split-process framework for fine-grained visual privacy control for augmented reality apps”, in “Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services”, MobiSys ’21, p. 14–27 (Association for Computing Machinery, New York, NY, USA, 2021a), URL <https://doi.org/10.1145/3458864.3467676>.
- Hu, J., A. Shaikh, A. Bahremand and R. LiKamWa, “Characterizing real-time dense point cloud capture and streaming on mobile devices”, in “Proceedings of the 3rd ACM Workshop on Hot Topics in Video Analytics and Intelligent Edges”, Hot-EdgeVideo ’21, p. 1–6 (Association for Computing Machinery, New York, NY, USA, 2021b), URL <https://doi.org/10.1145/3477083.3480155>.
- Hu, J., A. Shearer, S. Rajagopalan and R. LiKamWa, “Banner: An image sensor reconfiguration framework for seamless resolution-based tradeoffs”, in “Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services”, MobiSys ’19, p. 236–248 (Association for Computing Machinery, New York, NY, USA, 2019), URL <https://doi.org/10.1145/3307334.3326092>.
- Hu, J., J. Yang, V. Delhivala and R. LiKamWa, “Characterizing the reconfiguration latency of image sensor resolution on android devices”, in “Proceedings of the 19th International Workshop on Mobile Computing Systems & Applications”, Hot-Mobile ’18 (ACM, 2018), URL <http://doi.acm.org/10.1145/3177102.3177109>.
- Huang, A. ., “Betrusted: Improving security through physical partitioning”, *IEEE Pervasive Computing* **19**, 2, 13–20 (2020).
- Huang, J., O. Schranz, S. Bugiel and M. Backes, “The art of app compartmentalization: Compiler-based library privilege separation on stock android”, *CCS ’17*, p. 1037–1049 (Association for Computing Machinery, New York, NY, USA, 2017), URL <https://doi.org/10.1145/3133956.3134064>.
- HUSSAIN, U., “See in the dark with night vision app for iphone 12 and its lidar sensor”, <https://www.ithinkdiff.com/night-vision-apple-lidar-see-in-dark/> (2021).
- Ibrahim, R. M., “Camera”, URL <https://github.com/rizwankce/Camera/> (2019).
- Jana, S., A. Narayanan and V. Shmatikov, “A scanner darkly: Protecting user privacy from perceptual applications”, in “Proceedings of the 2013 IEEE Symposium on Security and Privacy”, SP ’13, pp. 349–363 (IEEE Computer Society, Washington, DC, USA, 2013), URL <https://doi.org/10.1109/SP.2013.31>.
- Jensen, J., J. Hu, A. Rahmati and R. LiKamWa, “Protecting visual information in augmented reality from malicious application developers”, *WearSys ’19*, p. 23–28 (Association for Computing Machinery, New York, NY, USA, 2019), URL <https://doi.org/10.1145/3325424.3329659>.

- Jiang, S., Z. Lin, Y. Li, Y. Shu and Y. Liu, *Flexible High-Resolution Object Detection on Edge Devices with Tunable Latency*, p. 559–572 (Association for Computing Machinery, New York, NY, USA, 2021), URL <https://doi.org/10.1145/3447993.3483274>.
- Jota, R., A. Ng, P. Dietz and D. Wigdor, “How fast is fast enough?: A study of the effects of latency in direct-touch pointing tasks”, in “Proceedings of the SIGCHI Conference on Human Factors in Computing Systems”, CHI ’13, pp. 2291–2300 (ACM, New York, NY, USA, 2013), URL <http://doi.acm.org/10.1145/2470654.2481317>.
- Kammerl, J., N. Blodow, R. B. Rusu, S. Gedikli, M. Beetz and E. Steinbach, “Real-time compression of point cloud streams”, in “2012 IEEE International Conference on Robotics and Automation”, pp. 778–785 (2012).
- Kodukula, V., S. B. Medapuram, B. Jones and R. LiKamWa, “A case for temperature-driven task migration to balance energy efficiency and image quality of vision processing workloads”, in “Proceedings of the 19th International Workshop on Mobile Computing Systems & Applications”, HotMobile ’18 (ACM, 2018), URL <http://doi.acm.org/10.1145/3177102.3177111>.
- Kodukula, V., A. Shearer, V. Nguyen, S. Lingutla, Y. Liu and R. LiKamWa, “Rhythmic pixel regions: Multi-resolution visual sensing system towards high-precision visual computing at low power”, in “Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems”, ASPLOS 2021, p. 573–586 (Association for Computing Machinery, New York, NY, USA, 2021), URL <https://doi.org/10.1145/3445814.3446737>.
- Lai, K., L. Bo, X. Ren and D. Fox, “Detection-based object labeling in 3d scenes”, in “2012 IEEE International Conference on Robotics and Automation”, pp. 1330–1337 (2012).
- Lampson, B. W., “A note on the confinement problem”, *Commun. ACM* **16**, 10, 613–615, URL <https://doi.org/10.1145/362375.362389> (1973).
- Lebeck, K., K. Ruth, T. Kohno and F. Roesner, “Securing augmented reality output”, in “2017 IEEE Symposium on Security and Privacy (SP)”, pp. 320–337 (2017).
- Lebeck, K., K. Ruth, T. Kohno and F. Roesner, “Arya: Operating system support for securely augmenting reality”, *IEEE Security Privacy* **16**, 1, 44–53 (2018).
- Lee, K., J. Yi, Y. Lee, S. Choi and Y. M. Kim, “Groot: A real-time streaming system of high-fidelity volumetric videos”, in “Proceedings of the 26th Annual International Conference on Mobile Computing and Networking”, MobiCom ’20 (Association for Computing Machinery, New York, NY, USA, 2020), URL <https://doi.org/10.1145/3372224.3419214>.
- LiKamWa, R., Y. Hou, J. Gao, M. Polansky and L. Zhong, “Redeye: Analog convnet image sensor architecture for continuous mobile vision”, in “Proceedings of the 43rd International Symposium on Computer Architecture”, ISCA ’16 (IEEE Press, 2016), URL <https://doi.org/10.1109/ISCA.2016.31>.

- Likamwa, R., J. Hu, V. Kodukula and Y. Liu, “Adaptive resolution-based tradeoffs for energy-efficient visual computing systems”, *IEEE Pervasive Computing* **20**, 2, 18–26 (2021).
- LiKamWa, R., B. Priyantha, M. Philipose, L. Zhong and P. Bahl, “Energy characterization and optimization of image sensing toward continuous mobile vision”, in “Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services”, *MobiSys ’13*, p. 69–82 (Association for Computing Machinery, New York, NY, USA, 2013a), URL <https://doi.org/10.1145/2462456.2464448>.
- LiKamWa, R., B. Priyantha, M. Philipose, L. Zhong and P. Bahl, “Energy characterization and optimization of image sensing toward continuous mobile vision”, in “Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services”, *MobiSys ’13* (ACM, 2013b), URL <http://doi.acm.org/10.1145/2462456.2464448>.
- LiKamWa, R. and L. Zhong, “Starfish: Efficient concurrency support for computer vision applications”, in “Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services, *MobiSys ’15*”, (ACM, 2015), URL <https://doi.org/10.1145/2742647.2742663>.
- Lin, F. X., Z. Wang, R. LiKamWa and L. Zhong, “Reflex: Using low-power processors in smartphones without knowing them”, *SIGPLAN Not.* **47**, 4, 13–24, URL <http://doi.acm.org/10.1145/2248487.2150979> (2012).
- Lin, T., P. Goyal, R. B. Girshick, K. He and P. Dollár, “Focal loss for dense object detection”, *CoRR* **abs/1708.02002**, URL <http://arxiv.org/abs/1708.02002> (2017).
- Liu, H., C. Li, G. Chen, G. Zhang, M. Kaess and H. Bao, “Robust keyframe-based dense SLAM with an RGB-D camera”, *CoRR* **abs/1711.05166**, URL <http://arxiv.org/abs/1711.05166> (2017).
- Liu, L. and M. Gruteser, “Edgesharing: Edge assisted real-time localization and object sharing in urban streets”, *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications* pp. 1–10 (2021).
- Liu, L., H. Li and M. Gruteser, “Edge assisted real-time object detection for mobile augmented reality”, in “The 25th Annual International Conference on Mobile Computing and Networking”, *MobiCom ’19* (Association for Computing Machinery, New York, NY, USA, 2019), URL <https://doi.org/10.1145/3300061.3300116>.
- Medina, F. P. and R. C. Paffenroth, “Machine learning in lidar 3d point clouds”, *CoRR* **abs/2101.09318**, URL <https://arxiv.org/abs/2101.09318> (2021).
- Micron Technology, “Calculating Memory System Power for DDR”, URL <https://www.micron.com/~/media/Documents/Products/Technical\%20Note/DRAM/TN4603.pdf> (2019).
- Microsoft Azure, “Azure Kinect DK”, <https://azure.microsoft.com/en-us/services/kinect-dk/> (2021).

- Nvidia, “nvjpeg library”, <https://docs.nvidia.com/cuda/nvjpeg/index.html> (2021).
- Nvidia developer, “Jetson TX2 Module”, <https://developer.nvidia.com/embedded/jetson-tx2> (2021).
- Olejnik, K., I. Dacosta, J. S. Machado, K. Huguenin, M. E. Khan and J. Hubaux, “Smarper: Context-aware and automatic runtime-permissions for mobile devices”, in “2017 IEEE Symposium on Security and Privacy (SP)”, pp. 1058–1076 (2017).
- ON Semiconductor, *AR0330 1/3-inch CMOS Digital Image Sensor*, rev. 18 (2017a).
- ON Semiconductor, *MT9P031 1/2.5-Inch 5 Mp CMOS Digital Image Sensor*, rev. 10 (2017b).
- Paine, J., “10 Real Use Cases for Augmented Reality: AR is set to have a big impact on major industries”, <https://www.inc.com/james-paine/10-real-use-cases-for-augmented-reality.html> (2020).
- PAINE, J., “10 real use cases for augmented reality”, URL <https://www.inc.com/james-paine/10-real-use-cases-for-augmented-reality.html> (2022).
- Point Cloud Library, “Point cloud library”, <https://pointclouds.org/> (2021).
- Prakash, S., A. Bahremand, L. D. Nguyen and R. LiKamWa, “Gleam: An illumination estimation framework for real-time photorealistic augmented reality on mobile devices”, in “Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services”, MobiSys ’19, pp. 142–154 (ACM, New York, NY, USA, 2019), URL <http://doi.acm.org/10.1145/3307334.3326098>.
- Priyantha, B., D. Lymberopoulos and J. Liu, “Littlerock: Enabling energy-efficient continuous sensing on mobile phones”, *IEEE Pervasive Computing* **10**, 2, 12–15 (2011).
- Qi, C. R., H. Su, K. Mo and L. J. Guibas, “Pointnet: Deep learning on point sets for 3d classification and segmentation”, *CoRR* **abs/1612.00593**, URL <http://arxiv.org/abs/1612.00593> (2016).
- Qian, F., B. Han, J. Pair and V. Gopalakrishnan, “Toward practical volumetric video streaming on commodity smartphones”, in “Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications”, HotMobile ’19, p. 135–140 (Association for Computing Machinery, New York, NY, USA, 2019), URL <https://doi.org/10.1145/3301293.3302358>.
- Qian, F., B. Han, Q. Xiao and V. Gopalakrishnan, “Flare: Practical viewport-adaptive 360-degree video streaming for mobile devices”, pp. 99–114 (2018).
- Qiu, H., F. Ahmad, F. Bai, M. Gruteser and R. Govindan, “Avr: Augmented vehicular reality”, in “Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services”, MobiSys ’18, p. 81–95 (Association for Computing Machinery, New York, NY, USA, 2018), URL <https://doi.org/10.1145/3210240.3210319>.

- Raval, N., A. Razeen, A. Machanavajjhala, L. P. Cox and A. Warfield, “Permissions plugins as android apps”, *MobiSys '19*, p. 180–192 (Association for Computing Machinery, New York, NY, USA, 2019), URL <https://doi.org/10.1145/3307334.3326095>.
- Raval, N., A. Srivastava, K. Lebeck, L. Cox and A. Machanavajjhala, “Markit: Privacy markers for protecting visual secrets”, in “Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct Publication”, *UbiComp '14 Adjunct*, pp. 1289–1295 (ACM, New York, NY, USA, 2014), URL <http://doi.acm.org/10.1145/2638728.2641707>.
- Raval, N., A. Srivastava, A. Razeen, K. Lebeck, A. Machanavajjhala and L. P. Cox, “What you mark is what apps see”, in “Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services”, *MobiSys '16*, pp. 249–261 (ACM, New York, NY, USA, 2016), URL <http://doi.acm.org/10.1145/2906388.2906405>.
- Reardon, J., Á. Feal, P. Wijesekera, A. E. B. On, N. Vallina-Rodriguez and S. Egelman, “50 ways to leak your data: An exploration of apps’ circumvention of the android permissions system”, in “28th USENIX Security Symposium (USENIX Security 19)”, pp. 603–620 (USENIX Association, Santa Clara, CA, 2019), URL <https://www.usenix.org/conference/usenixsecurity19/presentation/reardon>.
- Redmon, J. and A. Farhadi, “Yolov3: An incremental improvement”, *CoRR abs/1804.02767*, URL <http://arxiv.org/abs/1804.02767> (2018).
- Riegler, G., A. O. Ulusoy and A. Geiger, “Octnet: Learning deep 3d representations at high resolutions”, *CoRR abs/1611.05009*, URL <http://arxiv.org/abs/1611.05009> (2016).
- Ringer, T., D. Grossman and F. Roesner, “Audacious: User-driven access control with unmodified operating systems”, pp. 204–216 (2016).
- Roesner, F., D. Molnar, A. Moshchuk, T. Kohno and H. J. Wang, “World-driven access control for continuous sensing”, in “Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security”, *CCS '14*, pp. 1169–1181 (ACM, New York, NY, USA, 2014), URL <http://doi.acm.org/10.1145/2660267.2660319>.
- Roy, N., A. Misra, C. Julien, S. K. Das and J. Biswas, “An energy-efficient quality adaptive framework for multi-modal sensor context recognition”, in “2011 IEEE International Conference on Pervasive Computing and Communications (PerCom)”, (2011).
- Rusu, R. B. and S. Cousins, “3D is here: Point Cloud Library (PCL)”, in “IEEE International Conference on Robotics and Automation (ICRA)”, (Shanghai, China, 2011).
- Ruzicka, V. and F. Franchetti, “Fast and accurate object detection in high resolution 4k and 8k video using gpus”, *CoRR abs/1810.10551*, URL <http://arxiv.org/abs/1810.10551> (2018).

- Sanchez Vicarte, J. R., B. Schreiber, R. Paccagnella and C. W. Fletcher, “Game of threads: Enabling asynchronous poisoning attacks”, in “Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems”, ASPLOS ’20, p. 35–52 (Association for Computing Machinery, New York, NY, USA, 2020), URL <https://doi.org/10.1145/3373376.3378462>.
- Schmalstieg, D. and G. Hesina, “Distributed applications for collaborative augmented reality”, in “Proceedings IEEE Virtual Reality 2002”, pp. 59–66 (2002).
- Shaikh, A., L. Nguyen, A. Bahreman, H. Bartolomea, F. Liu, V. Nguyen, D. Anderson and R. LiKamWa, “Coordinate: A spreadsheet-programmable augmented reality framework for immersive map-based visualizations”, in “2019 IEEE International Conference on Artificial Intelligence and Virtual Reality (AIVR)”, pp. 134–1343 (2019).
- Shekhar, S., M. Dietz and D. S. Wallach, “Adsplit: Separating smartphone advertising from applications”, in “21st USENIX Security Symposium (USENIX Security 12)”, pp. 553–567 (USENIX Association, Bellevue, WA, 2012), URL <https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/shekhar>.
- Shen, Y., H. Tian, Y. Chen, K. Chen, R. Wang, Y. Xu, Y. Xia and S. Yan, “Occlum: Secure and efficient multitasking inside a single enclave of intel sgx”, in “Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems”, ASPLOS ’20, p. 955–970 (Association for Computing Machinery, New York, NY, USA, 2020), URL <https://doi.org/10.1145/3373376.3378469>.
- Simoens, P., Y. Xiao, P. Pillai, Z. Chen, K. Ha and M. Satyanarayanan, “Scalable crowd-sourcing of video from mobile devices”, in “Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services”, MobiSys ’13, p. 139–152 (Association for Computing Machinery, New York, NY, USA, 2013), URL <https://doi.org/10.1145/2462456.2464440>.
- Speciale, P., J. L. Schönberger, S. B. Kang, S. N. Sinha and M. Pollefeys, “Privacy preserving image-based localization”, in “2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)”, pp. 5488–5498 (2019).
- Square, Inc., “OkHttp”, <https://square.github.io/okhttp/> (2021).
- statista, “Forecast number of mobile devices worldwide from 2020 to 2025 (in billions)”, URL <https://www.statista.com/statistics/245501/multiple-mobile-device-ownership-worldwide/> (2022).
- The kernel development community, “ioctl vidioc\_reqbufs”, URL <https://linuxtv.org/downloads/v4l-dvb-apis/uapi/v4l/vidioc-reqbufs.html> (2019).
- The kernel development community, “11.1. file: media/v4l/capture.c”, URL <https://www.kernel.org/doc/html/v4.11/media/uapi/v4l/capture.c.html> (2021).

- Thomson, C., “Point clouds and vr: The future of point cloud visualisation”, <https://info.vercator.com/blog/point-clouds-and-vr-the-future-of-point-cloud-visualisation> (2021).
- tp-link, “TL-WR940N V6”, <https://www.tp-link.com/us/home-networking/wifi-router/tl-wr940n/> (2021).
- Track, F. and D. Kilpatrick, “Privman: A library for partitioning applications”, (2003).
- Unity, “Unity Metacast”, <https://unity.com/sports> (2021).
- Vuforia, “Innovate with industrial augmented reality”, URL <https://www.ptc.com/en/products/augmented-reality/> (2021).
- Wang, T., X. Zhu and D. Lin, “Reconfigurable voxels: A new representation for lidar-based point clouds”, CoRR **abs/2004.02724**, URL <https://arxiv.org/abs/2004.02724> (2020).
- Wang, X., A. Continella, Y. Yang, Y. He and S. Zhu, “Leakdoctor: Toward automatically diagnosing privacy leaks in mobile applications”, Proc. ACM Interact. Mob. Wearable Ubiquitous Technol. **3**, 1, URL <https://doi.org/10.1145/3314415> (2019).
- Wikipedia, “Ply (file format)”, [https://en.wikipedia.org/wiki/PLY\\_\(file\\_format\)](https://en.wikipedia.org/wiki/PLY_(file_format)) (2021).
- Xue, H., Y. Ju, C. Miao, Y. Wang, S. Wang, A. Zhang and L. Su, “Mmmesh: Towards 3d real-time dynamic human mesh construction using millimeter-wave”, in “Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services”, MobiSys ’21, p. 269–282 (Association for Computing Machinery, New York, NY, USA, 2021), URL <https://doi.org/10.1145/3458864.3467679>.
- Zhang, B., A. Davoodi and Y.-H. Hu, “Exploring energy and accuracy tradeoff in structure simplification of trained deep neural networks”, in “Proceedings of the 23rd Asia and South Pacific Design Automation Conference”, ASPDAC ’18 (IEEE Press, 2018), URL <http://dl.acm.org/citation.cfm?id=3201607.3201693>.
- Zhang, L. L., S. Han, J. Wei, N. Zheng, T. Cao, Y. Yang and Y. Liu, “Nn-meter: Towards accurate latency prediction of deep-learning model inference on diverse edge devices”, MobiSys ’21, p. 81–93 (Association for Computing Machinery, New York, NY, USA, 2021a), URL <https://doi.org/10.1145/3458864.3467882>.
- Zhang, W., Z. He, L. Liu, Z. Jia, Y. Liu, M. Gruteser, D. Raychaudhuri and Y. Zhang, “Elf: Accelerate high-resolution mobile deep vision with content-aware parallel offloading”, in “Proceedings of the 27th Annual International Conference on Mobile Computing and Networking”, MobiCom ’21, p. 201–214 (Association for Computing Machinery, New York, NY, USA, 2021b), URL <https://doi.org/10.1145/3447993.3448628>.



- Zhang, W., S. Li, L. Liu, Z. Jia, Y. Zhang and D. Raychaudhuri, “Hetero-edge: Orchestration of real-time vision applications on heterogeneous edge clouds”, in “IEEE INFOCOM 2019 - IEEE Conference on Computer Communications”, pp. 1270–1278 (2019).
- Zhang, X., A. Ahlawat and W. Du, “Aframe: Isolating advertisements from mobile applications in android”, in “Proceedings of the 29th Annual Computer Security Applications Conference”, ACSAC ’13, p. 9–18 (Association for Computing Machinery, New York, NY, USA, 2013), URL <https://doi.org/10.1145/2523649.2523652>.
- Zhang, X., A. Zhang, J. Sun, X. Zhu, Y. E. Guo, F. Qian and Z. M. Mao, “Emp: Edge-assisted multi-vehicle perception”, in “Proceedings of the 27th Annual International Conference on Mobile Computing and Networking”, MobiCom ’21, p. 545–558 (Association for Computing Machinery, New York, NY, USA, 2021c), URL <https://doi.org/10.1145/3447993.3483242>.
- Zhou, Q.-Y., J. Park and V. Koltun, “Open3D: A modern library for 3D data processing”, arXiv:1801.09847 (2018).
- Zhou, Y. and O. Tuzel, “Voxelnet: End-to-end learning for point cloud based 3d object detection”, CoRR [abs/1711.06396](https://arxiv.org/abs/1711.06396), URL <http://arxiv.org/abs/1711.06396> (2017).
- Zhu, X., J. Sun, X. Zhang, Y. E. Guo, F. Qian and Z. M. Mao, “Mpbond: Efficient network-level collaboration among personal mobile devices”, in “Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services”, MobiSys ’20, p. 364–376 (Association for Computing Machinery, New York, NY, USA, 2020), URL <https://doi.org/10.1145/3386901.3388943>.