How to Pipeline Frame Transfer and Server Inference in Edge-assisted AR to Optimize AR Task Accuracy?

Pranab Dash*
Purdue University

Z. Jonny Kong*
Purdue University

Y. Charlie Hu
Purdue University

Chris Turner
Charter Communications

Dell Wolfensparger
Charter Communications

Mun Gi Choi
Charter Communications

Abhinav Kshitij
Charter Communications

Viviane E. McLandrich
Charter Communications

ABSTRACT

Edge-assisted AR supports high-quality AR on resource-constrained mobile devices by offloading high-rate camera-captured frames to powerful GPU edge servers to perform heavy vision tasks. Since the result of an offloaded frame may not come back in the same frame interval, edge-assisted AR designs resort to local tracking on the last server returned result to generate more accurate result for the current frame. In such an offloading+local tracking paradigm, reducing the staleness of the last server returned result is critical to improving AR task accuracy.

In this paper, we present MPCP, an online offloading scheduling framework that minimizes the staleness of server-returned result in edge-assisted AR by optimally pipelining network transfer of frames to the edge server and the Deep Neural Network inference on the edge server. MPCP is based on model predictive control (MPC). Our evaluation results show that MPCP reduces the depth estimation error by up to 10.0% compared to several baseline schemes.

CCS CONCEPTS

• **Human-centered computing** → **Ubiquitous and mobile computing systems and tools**; **Ubiquitous and mobile computing design and evaluation methods**.

KEYWORDS

Edge-assisted Augmented Reality, Deep Neural Network, Edge Computing

ACM Reference Format:

Pranab Dash, Z. Jonny Kong, Y. Charlie Hu, Chris Turner, Dell Wolfensparger, Mun Gi Choi, Abhinav Kshitij, and Viviane E. McLandrich. 2023. How to Pipeline Frame Transfer and Server Inference in Edge-assisted AR to Optimize AR Task Accuracy? . In *6th International Workshop on Edge Systems, Analytics and Networking (EdgeSys '23), May 8, 2023, Rome, Italy*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3578354.3592870

*Both authors contributed equally to the paper.

EdgeSys '23, May 8, 2023, Rome, Italy

ACM ISBN 979-8-4007-0082-8/23/05.
https://doi.org/10.1145/3578354.3592870

1 INTRODUCTION

A complex Augmented Reality (AR) app often needs to perform a number of challenging tasks, e.g., pose estimation, object detection, and depth estimation, in order to understand and interact with the physical environment to provide the user with truly immersive experience. In recent years, Deep Neural Networks (DNN) models have been developed for performing these tasks with high accuracy. However, such DNN models are typically too computation-intensive to run on resource-constrained mobile devices in real time. As a result, edge-assisted AR, which offloads DNN inference to more powerful edge servers, has emerged as a popular approach to supporting high-quality AR (e.g., [8, 14, 17, 19, 27, 31]).

Since the result of an offloaded frame may not come back in the same frame interval, instead of simply reusing the last server returned result (for an older frame), edge-assisted AR designs (e.g., [7, 8, 19]) typically resort to performing local tracking to generate more accurate result for the current frame. Specifically, a local tracker runs on the mobile device and adjusts the DNN inference results for the last offloaded frame f_l sent back by the server to generate refined results for the current frame f_c, by analyzing the changes between the stale frame f_l and the current frame f_c.

A variety of local trackers have been developed for popular computer vision tasks. For example, object detection local trackers makes of feature extraction & matching [4, 8], correlation filters [20, 26] and optical flow [7], depth estimation local trackers are based on warping [21], and human pose estimation utilizes motion vectors [12, 19, 30]. However, local trackers cannot fully eliminate the impact of stale results, and their accuracy drops with the distance between f_l and f_c. For example, while object detection local trackers can follow existing objects in the frame, they typically cannot detect new objects. Therefore, in such an offloading+local tracking paradigm, reducing the staleness of the returned result directly translates into improved accuracy.

In this paper, we study how to apply pipelining, to minimize the staleness of the last server returned result used in local tracking in edge-assisted AR. We make a key observation that the intuitive solution of simply offloading the most recent frame just in time to keep the GPU server busy, which results in offloading the most frames, may not minimize the staleness of the last server returned result, due to an intricate discretization effect, i.e., server DNN inference time is often not an integer multiple of the frame interval (e.g., 33.3ms under 30 FPS).

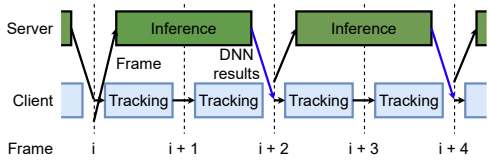


Figure 1: The offloading + tracking paradigm.

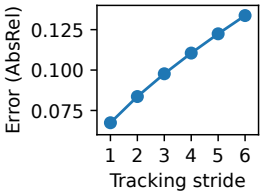


Figure 2: The local tracking error increases with tracking stride.

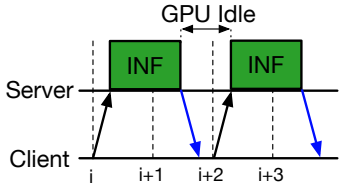


Figure 3: The GPU is idle between sending the result of the previous frame, and receiving the next frame.

We then present an online offloading scheduling algorithm called MPC-based pipelining (MPCP) that employs model predictive control [6] to optimally pipeline network transfer of frames to the edge server and DNN inference on the edge server to minimize the staleness of server-returned result in edge-assisted AR.

We have implemented MPCP in an Android app and evaluated it under Wi-Fi and emulated LTE and 5G networks based on recent measurements for two representative AR tasks, object detection and depth estimation, under a wide variety of videos generated from a photo-realistic simulator commonly used for autonomous driving, CARLA. Our evaluation results show that MPCP reduces the depth estimation errors by up to 10.0% over several baseline schemes, under different network conditions.

2 BACKGROUND

2.1 The Offloading + Tracking Paradigm

Due to the high accuracy of computational-heavy DNNs, offloading (also known as edge-assisted solutions) has been increasingly used to enable high-quality AR [8, 19, 31]. When offloading an AR task to the server, the offloaded result usually cannot return in the same frame interval (i.e. 33.3 ms for 30 FPS). The reason is two-fold. First, even with fast server GPUs, the inference latency of a typical DNN is tens of milliseconds. For example, the median inference latency of the 64 models in Meta's object detection model zoo [2] is 69.5ms on the Nvidia V100 GPU, and only 3% of the models run within 33.3ms. Second, the network transfer time and RTT latency also contribute to the total latency. The RTT can be a few milliseconds under Wi-Fi, 14-20 milliseconds under 5G mmWave, and 35-55 milliseconds under LTE [13]. Consequently, running under today's GPUs and networks, an offloaded frame's result typically returns several frames later, and hence the result for a past frame, i.e., the last server returned result, has to be used for the frames until the next result is returned from the server.

To mitigate the staleness of the returned results, *fast-tracking* has been commonly used by researchers [8, 19, 22] and practitioners [3, 29]. It refers to a family of lightweight algorithms that run locally on the device, take as input the result for the last offloaded frame f_l, and generate the result for the current frame f_c. A local tracker usually finishes within a frame time, as shown in Figure 1. For example, local trackers for object detection are usually based on feature matching [4, 8], optical flow [7], or motion vectors [19]; local trackers for depth estimation employ warping [21], a geometry-based algorithm.

2.2 The Need for Pipelining

While local tracking mitigates the staleness of the server-returned result for f_l and improves the accuracy of the current frame f_c, its accuracy is still inferior compared to the accuracy of running the DNN directly on f_c (denoted as offline accuracy, as it could not be done online), and the gap widens with the *tracking stride*, measured as the frame ID difference between f_l and f_c. We offline profiled the local tracker accuracy for the depth estimation task, on a dataset generated by the Carla simulator (see §4.1). The server runs a state-of-the-art AdaBins [5] DNN model, and the phone runs warping [21], a geometric-based algorithm, as the local tracker for depth estimation. Figure 2 shows that the absolute relative error between the tracked and ground truth depth map increases by 0.0133 per tracking stride increment on average.

The offloaded frame f_l will return after an end-to-end (E2E) latency of T_{e2e} (e.g., L frame intervals), and be used by the local tracker to produce results starting from frame $f_l + L$, to frame $f_l + L + k - 1$, after which the results for the next offloaded frame f_{l+k} returns. In other words, the average tracking stride will be (L+L+1+...+L+k-1) / k = L + (k-1)/2. This suggests that there are two ways to reduce the average tracking stride. First, reducing the E2E offloading latency T_{e2e} reduces L and hence the average tracking stride. Second, when not every frame can be offloaded (which is usually the case since the DNN runtimes exceed one frame time), reducing the interval between two consecutively offloaded frames k also reduces the average tracking stride.

Related Works. Reducing the E2E offloading latency has been extensively studied. Existing approaches include employing image or video encoding on the offloaded frames [8, 19, 22], splitting the DNN between the client and the server and offloading the intermediate representations [11, 16, 17, 27], dividing a frame into segments to pipeline the frame transmission with DNN inference [19, 25], and identifying regions of interests (RoI) and encoding them with a higher quality [10, 19, 31].

There has been little work on reducing the interval between consecutively offloaded frames. Existing work determines when to offload a new frame following one of the following policies. (1) Offload a frame when detecting significant changes in the frame content [8, 19], which conserves bandwidth usage but will result in low accuracy due to higher tracking strides. (2) Offload a frame as soon as the result of the previous offloaded frame has returned [19, 22], which prevents queue buildup in the network or at the server GPU. However, since the offloading of the next frame happens after receiving the result for the previous frame, the server GPU will be idle between sending the result for the previous frame and waiting to receive the next uploaded frame and hence under-utilized, as shown in Figure 3.

In this paper, we study how to apply a classic latency-hiding technique, pipelining the network transfer and server computation, in the offloading+tracking paradigm for AR, to improve the accuracy of the offloaded tasks.

2.3 Challenges

Designing a framework that optimally pipelines network transfer and GPU inference faces several unique challenges.

The pipelining scheduling needs to adapt to network dynamics online. Pipelining the network transfer and DNN inference depends on the runtime of each. While the DNN inference latency on the server is generally stable, the client's network condition could vary. Today's highest-performance network, 5G mmWave, exhibits fast-changing bandwidth due to user movement [24], which results in varying uplink and downlink transfer latencies. The user also experiences vertical handoffs which contribute to varying network latency [15]. Therefore, it is not feasible to decide on a static pipelining schedule offline that would perform well online. Rather, the schedule needs to be dynamically adapted online, according to the network dynamics. This requires a pipelining scheduling algorithm that can run in real-time.

Always eliminating GPU idle interval may not be beneficial, since it may require sending stale frames. A strawman design is to perform pipelining such that the server GPU is always kept busy, by ensuring that there is always a frame uploaded upon the completion of the previous DNN inference. The intuition is that this design makes the GPU process more frames, and hence reduces the average interval between two consecutively offloaded frames. We refer to this design as Perfect Pipelining (**PP**). However, a detailed examination shows that in the unique context of edge-assisted AR which has a constant frame arrival rate, PP does not always result in the lowest tracking stride. This is because the inference latency is usually not a multiple of frame interval (e.g., 33.3 ms under 30 FPS). In such cases, PP often has to offload a frame in the middle of a frame interval, so that this frame arrives at the server right before the GPU inference of the previous offloaded frame finishes. For example, when the inference latency is 1.2 frame times in Figure 4, offloaded frame $i+1$ arrives at the server at $t = 1.2$ (unit in frame times), frame $i+2$ arrives at $t = 2.4$, frame $i+3$ arrives at $t = 3.6$, i.e., frame $i + 3$ is already 0.6 unit stale when it arrives at the server.

To address this stale-frame issue, an alternative design is to stall the pipeline by skipping the current frame and waiting for and sending the next frame. We refer to this design as Pipelining with Stall (**PS**). As shown in Figure 4, frames in PS are always sent once they become available; they are 0.3 units stale when arriving at the server (assuming uploading takes 0.3 frame interval time.)

However, we observe whether PS or PP will result in a lower *average* tracking stride is far from obvious; as the tracking stride is affected by both the offloading frequency and staleness of offloaded frames in intricate ways, which in turn depend on the upload/download latency and the DNN inference latency, relative to the frame interval time. We illustrate this using an example shown in Figure 5. In Figure 5a we assume the uplink and the downlink latencies (which include the RTT and transmission latency) are both 0.3 frame times, and the DNN inference latency is 1.2 frame

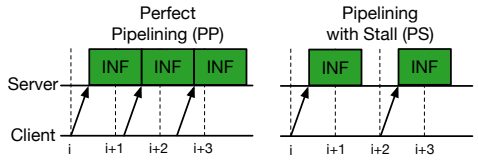


Figure 4: PP utilizes GPU in full but may send a stale frame (in the middle of a frame interval). PS always sends a fresh frame but utilizes less GPU and thus may offload fewer frames.

times. PS achieves a lower average stride of 2.50 by injecting stalls in the pipeline, compared with 2.74 for PP. However, as shown in Figure 5b, when the uplink and the downlink latencies increase from 0.3 to 0.5 frame times, PS instead achieves a higher stride of 3.50 as compared to PP's 3.26.

In essence, each of PP and PS excels in one metric but falls short in the other. As shown in Figure 4, offloading in PP keeps the server GPU busy and thus has the desirable effect of offloading more frames, but has the undesirable effect of offloading a relatively stale frame to the GPU server, while the opposite is true for PS. Maximizing the benefits of pipelining the network transfer and GPU inference requires carefully balancing these two conflicting factors. This suggests that the optimal offloading schedule for a given scenario is to combine PP and PS at the frame granularity. Going back to our example in Figure 5, the optimal schedule (labeled as Oracle) strategically adapts between sending an older frame to keep the GPU busy and stalling the pipeline to send the next, more recent frame.

No analytical or algorithmic solution. Due to the intricate discretization effect discussed above, i.e., DNN inference latency is not a multiple of a frame interval, it is difficult to develop a closed-form formula that takes as input the DNN inference latency, upload/download time, and frame interval time, and outputs an optimal offloading schedule. Furthermore, the problem cannot be efficiently solved with dynamic programming (DP) as it doesn't have the optimal substructure property[1].

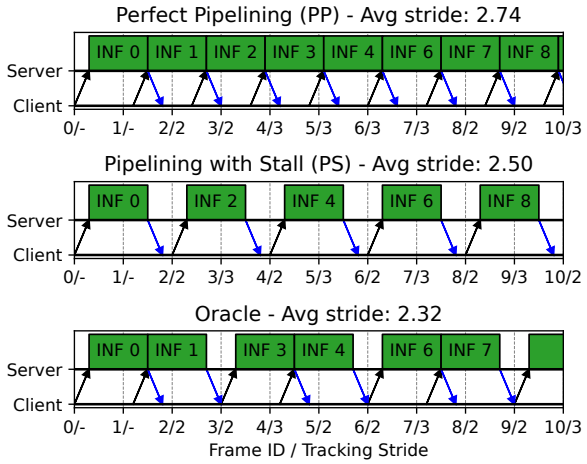
3 DESIGN

We design MPC-based pipelining (MPCP), a pipelining scheduling framework for the offloading + tracking paradigm with the goal of achieving the lowest average tracking stride. The framework is adaptive to varying network conditions and extensible to DNN models for different tasks.

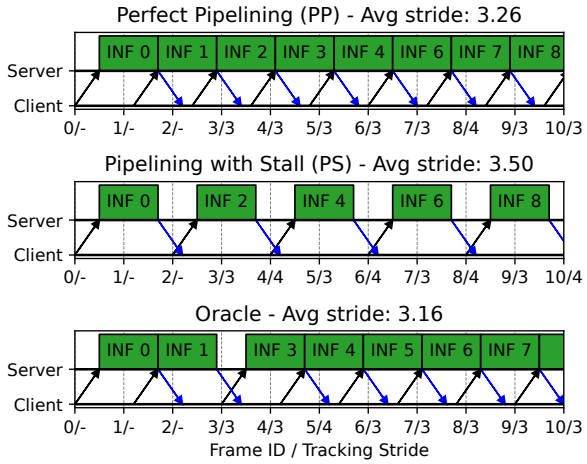
3.1 Overview

We design our new scheduler using model predictive control (MPC) [6, 28], a control theory-based technique. Given the model for a system, MPC produces steps of actions that optimize the system over a finite time-horizon, but only executes the first one or few steps, and then optimizes again. This allows it to optimize not only the current step but also future steps. MPC does not have a learned component, and thus is naturally extensible to varying DNN models and network conditions online, to meet our extensibility goals.

[1]The optimal schedule up to frame i, denoted as $S[i]$, cannot be derived from $S[i-k]$ for some k, because the choice of $S[i-k]$ affects the server GPU state up to a certain number of frames after frame $i-k$ and thus cannot be solved independently.



(a) UL latency 0.3, DL latency 0.3, INF latency 1.2 (in frame times).



(b) UL latency 0.5, DL latency 0.5, INF latency 1.2 (in frame times).

Figure 5: The pipelining timeline and the average tracking stride (over 20 frames, but only first 10 frames shown). The x axis shows the [Frame ID / tracking stride] pair for each frame, e.g., 3/2 in the PP figure on the left means frame 3 has a tracking stride 2, because it performs tracking on the server returned result for frame 1.

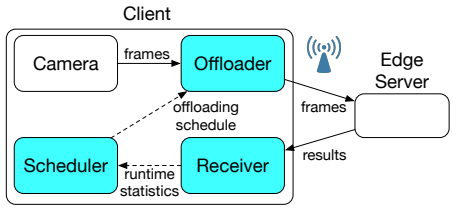


Figure 6: The MPCP offloading framework.

Figure 6 shows the workflow of MPCP. The *Scheduler* runs periodically and produces an offloading schedule. Following this schedule, the *Offloader* offloads the frames captured from the camera to the server. The server-returned result is received by the *Receiver*, which passes the results to the upper-layer application, and also collects the runtimes of uplink/downlink transfer (which includes RTT and transmission time), and DNN inference. These runtimes are measured by synchronizing the clocks of the client and server following a similar mechanism to NTP [23], and having the server continuously report the needed timestamps back to the client. The client maintains a moving average of runtimes to adapt to dynamic variations. The measured runtimes are passed to the *Scheduler*, which is described below.

3.2 The MPCP Scheduler

Schedule representation. We define a schedule for a horizon of n frames as a boolean vector $s = \{b_0, b_1, ..., b_{n-1}\}$, where $b_i \in \{0, 1\}$. $b_i = 1$ means the i-th frame will be offloaded, and $b_i = 0$ otherwise. Since the server performs inference on one frame at a time, and the subsequent frames will queue up if the server GPU is busy, offloading a frame in the middle of a frame interval (e.g., frame 3 at $t = 3.5$) and arriving at the server right before the previous inference finishes is equivalent to offloading a frame as soon as it becomes available (e.g., frame 3 at $t = 3.0$) and arriving at the server before the previous inference finishes. Therefore, for each frame that *needs* to be offloaded, the *Offloader* offloads it as soon as it becomes available. Hence, a binary vector is a complete representation of the entire schedule space.

Search space pruning. Recall that MPC works by finding the best schedule in a finite horizon. For a horizon of n frames, a naive enumeration of the possible schedules gives a search space of $O(2^n)$, which quickly becomes intractable once n exceeds tens of frames. When running an AR app at 30 FPS, a 1-second horizon translates to 30 frames and a search space size of $2^{30} = 1073741824$. To tackle this problem, we prune the search space as follows.

First, we apply a greedy heuristic to prune schedules that will cause significant congestion or under-utilization of the GPU. With DNN inference time of t_{inf} frame times, schedules that offload two frames less than $\lfloor t_{inf} \rfloor$ frames apart at any point, or more than $\lceil t_{inf} \rceil$frames apart at any point, are sub-optimal in terms of tracking stride. This is because offloading two frames (i, j) less than $\lfloor t_{inf} \rfloor$ apart will cause frame j to queue at least one frame time on the GPU, so it is sub-optimal to offloading frames $(i, j+1)$. Similarly, offloading frames (i, j) more than $\lceil t_{inf} \rceil$ apart will cause an idle period of more than one frame time on the GPU before inference on j begins, so it is sub-optimal to offloading frames $(i, j-1)$. After removing these sub-optimal schedules, Table 1 shows the size of the remaining search space for a 30-frame horizon is reduced significantly. Note that the search space is the same as long as the t_{inf} falls under the same frame time bin.

While the above greedy heuristic significantly cuts the search space, the search space is still large when the DNN inference time is small. For example, when the inference time is between 1-2 frame times, the search space is 1346269. This is because a faster DNN enables more frames to be offloaded within a fixed horizon, thus inflating the search space. By plotting the CDF of average tracking strides for the schedules in the search space, shown in Figure 7, we observe that the average stride distribution does not have a long tail for small strides. In other words, there exist many schedules that

Inference duration	Search space
0-1	1
1-2	1346269
2-3	3329
3-4	285
4-5	71

Table 1: The scheduler search space size after the greedy-based pruning heuristic, for a horizon of 30 frames.

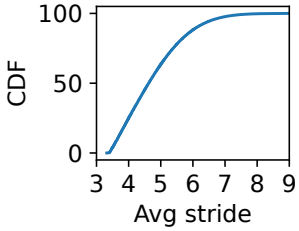


Figure 7: CDF of average strides of the 1346269 schedules in the search space, when the DNN inference time is 1.5 frame times.

give average tracking strides close to that of the optimal schedule. Therefore, we cap the search space at N by randomly sampling up to N candidates, with a high probability some of these candidates will be close to optimal. In practice, we set $N = 2000$ to achieve real-time performance.

Selecting the best schedule. Given the uplink time, DNN inference time, and downlink time, we calculate when each offloaded frame will return and then obtain the tracking stride for the frames in the horizon. We calculate the average tracking stride for each candidate schedule and select the schedule with the lowest tracking stride.

4 EVALUATION

4.1 Methodology

Implementation. We implemented the MPCP pipelining framework in an Android app and the edge server in Python. The client produces frames at 30 FPS, and performs H264 encoding on the offloaded frames using the hardware encoder provided by Android MediaCodec API [1]. The scheduler runs every 500 ms, optimizing for a horizon of 1 second. We used warping [21], a lightweight geometry-based algorithm, as the local tracker for depth estimation.

We implemented the server in about 600 lines of Python code with the PyTorch framework. We chose AdaBins [5] as the DNN model for depth estimation. The server decodes incoming frames with ffmpeg, and performs DNN inference on each frame sequentially. The subsequent frames will be queued until the inference on the current frame finishes. To efficiently transmit the depth maps to the client, the server downsizes them from 832×256 to 416×128 and quantizes the depth map to `uint16`.

Evaluation setup. We run the client Android app on a Pixel 5 phone, which has a Qualcomm 765G SoC that is equipped with an Octa-core Kryo 475 CPU and an Adreno 620 GPU. Our server is equipped with an NVIDIA A40 GPU. We evaluate the framework under 802.11ac, LTE, and 5G mmWave networks. For experiments under 802.11ac, we connect the phone to an 802.11ac access point, which has a wired connection to the server. For experiments under LTE and 5G mmWave, we emulate the network latency and throughput using the Linux tc tool, based on recent measurements [13] (35 ms RTT, 53 Mbps average throughput for LTE, 14 ms and 150 Mbps average for 5G mmWave).

Datasets. We used CARLA [9], a photo-realistic simulator commonly used for autonomous driving, to generate synthetic datasets

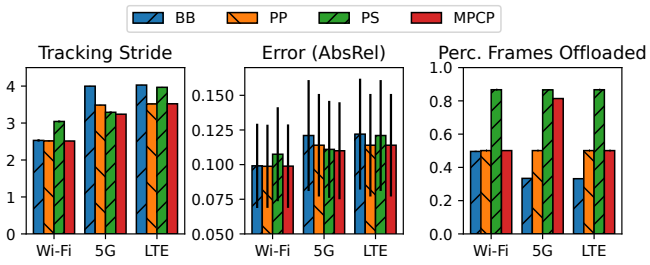


Figure 8: Results for offloading depth estimation.

with ground truth labels for evaluation. We generated 30 videos each 40 seconds long where the camera is mounted at the top of a vehicle, capturing frames at 30 FPS with the resolution of 832×256. We re-trained the depth estimation model using 20 of the videos and the rest are used for evaluation.

Metrics. We report the average tracking stride, i.e., the staleness of the last server-returned results, as well as the resulting accuracies of individual AR tasks using commonly used task-specific accuracy metrics. For the depth estimation task, we calculate the absolute relative error (AbsRel) between the estimated depth map and the ground truth depth map. We also report the number of frames offloaded which reveals the offloading behaviors of the schedulers differ. We treat the first 5 seconds of each run as the ramp-up phase and remove them when reporting the results.

4.2 Baselines

While there are many works on offloading to an edge server, they are often evaluated on a per-frame basis on a non-AR setting [11, 17, 27, 31], or assume the GPU is not the bottleneck [10, 14, 16, 18], and thus does not involve a scheduler. Among the works that require a scheduler, some offload after significant changes are detected [8] which will inevitably result in high tracking stride and low accuracy, or offload the new frame if there is no outstanding request [19, 21, 22], which we include as one of our baselines. Additionally, we include the two baselines described in §2.3. In summary, we compare with the following baselines:

Back-to-back (BB). When a new frame arrives, it is offloaded if there is no outstanding offloaded request. This was used in [19, 21, 22].

Perfect Pipelining (PP). This setup was described in §2.3, which pipelines the frame offloading with the server GPU inference such that the GPU is kept busy.

Pipelining with Stall (PS). This setup was described in §2.3; when a new frame arrives, it is offloaded if that does not cause queuing at the server.

4.3 Results

We evaluate the three baseline schedulers and the MPCP scheduler in offloading the depth estimation task under varying network conditions including Wi-Fi, 5G mmWave, and LTE. The DNN inference latency on the NVIDIA A40 GPU is 40.0 ms. The results for the schedulers are shown in Figure 8. We make the following observations:

Tracking stride and percent of frames offloaded. *BB scheduler is vulnerable to sub-optimal network conditions.* The BB scheduler achieves tracking stride of 2.53 under Wi-Fi which is close

to the best scheduler. However, under 5G and LTE it becomes the worst scheduler, achieving tracking strides of 3.99 and 4.03, respectively. This is because BB does not pipeline network transfer and DNN inference, and thus is more vulnerable to high network latencies. This is also evident in BB offloading fewer frames (33%, or 1 every 3) than the other schedulers under 5G or LTE, because the E2E offloading latency is between 2-3 frame times (2.26 and 2.96 respectively).

PP or PS scheduler performs well under selected network conditions. PP achieves tracking strides of 2.53 and 3.52 under Wi-Fi and LTE respectively which are similar to the best scheduler, but achieves a tracking stride of 3.49 under 5G mmWave which is 0.25 higher than the best scheduler. In contrast, PS performs similarly as the best scheduler under 5G mmWave, achieving a tracking stride of 3.29, but performs significantly worse under Wi-Fi and LTE, achieving tracking strides of 3.04 and 3.97 respectively, which are 0.53 and 0.45 higher. This shows that whether PS or PP is better is not obvious and depends on the network transfer time. In terms of the percent of frames offloaded, PP always offloads about 87% of the frames under all network conditions, because it follows a pipeline that keeps the GPU busy, whereas PS offloads around 50% of the frames.

The MPCP scheduler achieves the lowest tracking stride in all network conditions. MPCP achieves tracking strides of 2.51, 3.24, and 3.52, under Wi-Fi, LTE, and 5G mmWave respectively. Furthermore, MPCP offloads fewer frames than PP but more frames than PS, showing that it is able to offload more frames when needed to strike a balance between offloading more frames to keep the GPU busy and stalling the pipeline to send the next frame to avoid sending stale frames.

Application accuracy. Shorter tracking strides translate into lower depth estimation errors. Across different network conditions, MPCP reduces the absolute relative error (in absolute) by 0.02% − 1.10% compared to BB, 0.00% − 0.38% compared to PP, and 0.08% − 0.87% compared to PS, which are considered significant in the machine learning community [5]. In relative terms, these translate to 0.3% − 10.0%, 0.0% − 3.4%, and 0.8% − 8.8% error reductions. Note that the standard deviation of the accuracy is high due to the content variation across videos. On the other hand, the tracking stride and percent of frames offloaded show small standard deviations because they are not affected by video content.

Scheduling overhead. With DNNs of inference latency between 1 and 2 frame times (where the MPC search space is the largest, as shown in Table 1), each run of the MPCP scheduler which happens every 500 ms takes only 3-7 ms, which is well below one frame time under 30 or 60 FPS.

5 CONCLUSION

With faster networks and larger DNNs, solutions for high-quality AR are moving towards edge-assisted designs. We presented an online offloading scheduling algorithm for edge-assisted AR that minimizes the staleness of server-returned result which translates into best accuracy of the offloaded AR task. Our scheduler exploits model predictive control to optimally pipeline network transfer of frames to the edge server and the DNN inference on the edge server. Evaluation of our prototype implementation on Android phones shows that MPCP reduces the errors of edge-assisted depth estimation by up to 10.0% over several popular baseline schemes under today's mobile networks.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful comments. This project is supported in part by Charter Communications and by NSF grant 2112778-CNS.

REFERENCES

[1] 2022. Android MediaCodec. https://developer.android.com/reference/android/media/MediaCodec.html.
[2] 2022. Detectron2. https://github.com/facebookresearch/detectron2/blob/main/MODEL_ZOO.md.
[3] A. Ahmadyan et al. 2020. Real-Time 3D Object Detection on Mobile Devices with MediaPipe. https://ai.googleblog.com/2020/03/real-time-3d-object-detection-on-mobile.html
[4] K. Apicharttrisorn et al. 2019. Frugal following: Power thrifty object detection and tracking for mobile augmented reality. In *Proc. of ACM SenSys.*
[5] S. Bhat et al. 2021. Adabins: Depth estimation using adaptive bins. In *Proc. of IEEE CVPR.*
[6] E.F. Camacho and C.B. Alba. 2013. *Model Predictive Control.* Springer London. https://books.google.com/books?id=tXZDAAAAQBAJ
[7] K. Chen et al. 2018. Marvel: Enabling mobile augmented reality with low energy and low latency. In *Proc. of ACM SenSys.*
[8] T. Chen et al. 2015. Glimpse: Continuous, real-time object recognition on mobile devices. In *Proc. of ACM SenSys.*
[9] A. Dosovitskiy et al. 2017. CARLA: An open urban driving simulator. In *Proc. of CoRL.*
[10] K. Du et al. 2020. Server-driven video streaming for deep learning inference. In *Proc. of ACM SIGCOMM.*
[11] A. Eshratifar et al. 2019. JointDNN: An efficient training and inference engine for intelligent mobile cloud computing services. *IEEE TMC* 20, 2 (2019), 565–576.
[12] Chengsi Gao et al. 2021. An intelligent video processing architecture for edge-cloud video streaming. In *Proc. of ACM/IEEE DAC.*
[13] M. Ghoshal et al. 2022. An in-depth study of uplink performance of 5G mmWave networks. In *Proc. of ACM SIGCOMM 5G-MeMU Workshop.*
[14] S. Han et al. 2016. MCDNN: An approximation-based execution framework for deep stream processing under resource constraints. In *Proc. of ACM MobiSys.*
[15] A. Hassan et al. 2022. Vivisecting mobility management in 5G cellular networks. In *Proc. of the ACM SIGCOMM.*
[16] C. Hu et al. 2019. Dynamic adaptive DNN surgery for inference acceleration on the edge. In *Proc. of IEEE INFOCOM.*
[17] Y. Kang et al. 2017. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In *Proc. of ACM ASPLOS.*
[18] Y. Li et al. 2020. Reducto: On-camera filtering for resource-efficient real-time video analytics. In *Proc. of ACM SIGCOMM.*
[19] L. Liu et al. 2019. Edge assisted real-time object detection for mobile augmented reality. In *Proc. of ACM MobiCom.*
[20] Ruoyang Liu, Lu Zhang, Jingyu Wang, Huazhong Yang, and Yongpan Liu. 2021. PETRI: Reducing Bandwidth Requirement in Smart Surveillance by Edge-Cloud Collaborative Adaptive Frame Clustering and Pipelined Bidirectional Tracking. In *2021 58th ACM/IEEE Design Automation Conference (DAC).* IEEE, 421–426.
[21] J. Meng et al. 2021. Do Larger (More Accurate) Deep Neural Network Models Help in Edge-assisted Augmented Reality?. In *Proc. of ACM SIGCOMM NAI Workshop.*
[22] J. Meng et al. 2022. Do we need sophisticated system design for edge-assisted augmented reality?. In *Proc. of ACM EdgeSys.*
[23] D. Mills. 1985. *Network time protocol (NTP).* Technical Report.
[24] A. Narayanan et al. 2022. A comparative measurement study of commercial 5G mmWave deployments. In *Proc. of IEEE INFOCOM.*
[25] X. Wang et al. 2021. Edgeduet: Tiling small object detection for edge assisted autonomous mobile vision. In *Proc. of IEEE INFOCOM.*
[26] Ran Xu et al. 2020. ApproxDet: content and contention-aware approximate object detection for mobiles. In *Proceedings of ACM SenSys.*
[27] S. Yao et al. 2020. Deep compressive offloading: Speeding up neural network inference by trading edge computation for network latency. In *Proc. of ACM SenSys.*
[28] X. Yin et al. 2015. A control-theoretic approach for dynamic adaptive video streaming over HTTP. In *Proc. of ACM SIGCOMM.*
[29] M. Yong. 2019. *Object Detection and Tracking using MediaPipe.* https://developers.googleblog.com/2019/12/object-detection-and-tracking-using-mediapipe.html
[30] Jinrui Zhang et al. 2020. MobiPose: Real-time multi-person pose estimation on mobile devices. In *Proc. of ACM SenSys.*
[31] W. Zhang et al. 2021. Elf: accelerate high-resolution mobile deep vision with content-aware parallel offloading. In *Proc. of ACM MobiCom.*