Appendix for Towards Streaming Perception

A Benchmark Details . . . 2
A.1 Additional Discussion on the Benchmark Definition . . . 2
A.2 Pseudo Ground Truth . . . 2
A.3 Simulation . . . 3
A.4 Dataset Annotation and Comparison . . . 5
A.5 Experiment Settings . . . 5
A.6 Alternate Task: Instance Segmentation . . . 6
A.7 Alternate Hardware: Tesla V100 . . . 7
B Solution Details . . . 10
B.1 Dynamic Scheduling . . . 10
B.2 Additional Details for Forecasting . . . 14
B.3 Additional Details for Visual Tracking . . . 16
B.4 Evaluation of Our Meta-Detector *Streamer* . . . 16
B.5 Implementation Details . . . 19
C Additional Baselines . . . 19
C.1 Forecasting Baselines . . . 19
C.2 An End-to-End Baseline . . . 21

We summary the contents of the appendix as follows. Appendix A describes additional details of our meta-benchmark, including discussion on the definition, pseudo ground-truth, simulation, dataset and instantiations for novel hardware and task. Appendix B provides additional details of our proposed solutions, including scheduling, tracking and forecasting. Finally, Appendix C includes additional baselines for a more thorough evaluation.

A Benchmark Details

A.1 Additional Discussion on the Benchmark Definition

In Section 3.1 (main text), we defined our benchmark as evaluation over a discrete set of frames. One might point out that a continuous definition is more consistent with the notion of estimating the state of the world at all time instants for streaming perception. First, we note that it is possible to define a continuous-time counterpart, where the ground truth can be obtained via polynomial interpolation and the algorithm prediction can be represented as a function of time (e.g., simply derived from extrapolating the discrete output). Also in Eq 4 (main text), the aggregation function (implicit in L) could be integration. However, our choice of a discrete definition is mainly for two reasons: (1) we believe a high-frame-rate data stream is able to approximate the continuous evaluation; (2) most existing single-frame metrics (L, e.g., average-precision) is defined with a discrete set of input and we prefer that our streaming metric is compatible with these existing metrics.

A.2 Pseudo Ground Truth

We use manually obtained ground-truth for bounding-box-based object detection. As we point out in the main text, one could make use of pseudo ground truth by simply running an (expensive but accurate) off-line detector to generate detections that could be used to evaluate on-line streaming detectors.

Here, we analyze the effectiveness of pseudo ground truth detection as a proxy for ground-truth. We adopt the state-of-the-art detector — Hybrid Task Cascade (HTC) [5] for computing the offline pseudo ground truth. As shown in Table 1 (main text), this offline detector dramatically outperforms all real-time streaming methods by a large margin. As shown in the main text, pseudo-streaming AP correlates extraordinarily well with ground-truth-streaming AP, with a normalized correlation coefficient of 0.9925. This suggests that pseudo ground truth can be used to rank streaming perception algorithms.

We emphasize that since we have constructed ArgoverseHD by deliberately annotating high frame rate bounding boxes, *we use real ground truth for evaluating detection performance*. However, obtaining such high-frame-rate annotations for instance segmentation is expensive. Hence we make use of pseudo ground-truth instance masks (provided by HTC) to benchmark streaming instance segmentation (Section A.6).

A.3 Simulation

In true hardware-in-the-loop benchmarking, the output timestamp s_j is simply the wall-clock time at which an algorithm produces an output. While we hold this as the gold-standard, one can dramatically simplify benchmarking by making use of simulation, where s_j is computed using runtimes of different modules. For example, s_j for a single-frame detector on a single GPU can be simulated by adding its runtime to the time when it starts processing a frame. Complicated perception stacks require considering runtimes of all modules (we model those that contribute > 1 ms) in order to accurately simulate timestamps.

Modeling runtime distribution Existing latency analysis [18,16,14] usually reports only the mean runtime of an algorithm. However, empirical runtimes are in fact *stochastic* (Fig. A), due to the underlying operating system scheduling and even due to the algorithm itself (e.g., proposal-based detectors often take longer when processing a scene with many objects). Because scene-complexity is often correlated across time, runtimes will also be correlated (a long runtime for a given frame may also hold for the next frame).

We performed a statistical analysis of runtimes, and found that a *marginal* empirical distribution to work well. We first run the algorithm over the entire dataset to get the empirical distribution of runtimes. At test time, we randomly sample a runtime when needed from the empirical distribution, without considering the correlation across time. Empirically, we found that the results (streaming AP) from a simulated run is within the variance of a real run.

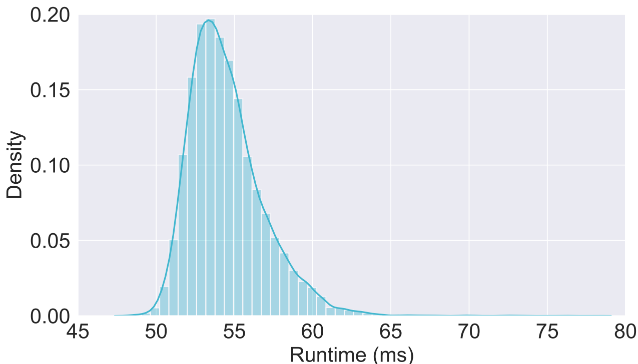

Fig. A. Runtime distribution for an object detector. Note that runtime is not constant, and this variance needs to be modeled in a simulation. This plot is obtained by running RetinaNet (ResNet 50) [14] on Argoverse 1.1 [4] with input scale 0.5.

Simulation for non-existent hardware/algorithm Through simulation, our evaluation protocol does not directly depend on hardware, but on a collection

of runtime distributions for different modules (known as a *runtime profile*). One thus has the freedom to alter the distributions. For example, we can simulate a faster algorithm simply by scaling down the runtime profile. Table 3 (main text), uses simulation to evaluate the streaming performance of a non-existent tracker that runs twice as fast as the actual implementation on-hand. The reduced runtime could have arisen from better hardware; one can run the benchmark on a Geforce GTX 1080 Ti GPU and simulate the performance on a Tesla V100 GPU. We find that Tesla V100 makes our detectors run 16% faster, implying we can scale runtime profiles accordingly. For example, Mask R-CNN R50 @ s0.5 produces a simulated-streaming AP of 12.652 while the real-streaming AP (on a V100) is 12.645, suggesting that effectivness of simulated benchmarking.

Infinite GPUs In simulation, we are not restricted by the number of physical GPUs present in a system. Therefore, we are able to perform analysis in the infinite GPU setting. In this setting, each detector or visual tracker runs on a different device without any interference with each other. Equivalently, we run a new GPU job on an existing device as long as it is idle. As a result, the simulation also provides information on how many GPUs are required for a particular infinite GPU experiment in practice (*i.e.*, the maximum number of concurrent jobs). We summarize the number of GPUs required for the experiments in the main text in Table A. This implies that our streaming benchmark can be used to inform hardware design of *future* robotic platforms.

Table A. Summary of the experiments in the infinite GPU settings (in the main text) and the number of GPUs needed in practice to achieve this performance (*i.e.*, the maximum number of concurrent jobs). This suggest that our simulation can also identify the optimal hardware configuration

Method	# of GPUs
Det (Table 1, row 8)	4
Det + Associate + Forecast (Table 2, row 3)	4
Det + Visual Track (Table 3, row 4)	9
Det + Visual Track + Forecast (Table 3, row 5)	9

Runtime-induced variance As mentioned in the previous section, runtime is stochastic and has a variance up to 11.1% (standard deviation normalized by mean). Fortunately, such a variance does not transfer to the variance of our streaming metric. Empirically, we found that the variance of streaming AP of different runs (by varying the random seed) is around 0.5% for the same algorithm. In comparison, independent training runs of Mask R-CNN [12] on MS COCO [15] with the *same random seed* yield a variance of 0.3% on the AP (cudnn back-propagation is stochastic by default) [13]. Since the stochastic noise of streaming evaluation is at the same scale as CNN training, we ignore runtime-induced variance for our evaluation.

A.4 Dataset Annotation and Comparison

Based on the publicly available video dataset Argoverse 1.1 [4], we build our dataset with high-frame-rate annotations for streaming evaluation — Argoverse-HD (High-frame-rate Detection). One key feature is that the annotation follows MS COCO [15] standards, thus allowing direct evaluation of COCO pre-trained models on this self-driving vehicle dataset. The annotation is done at 30 FPS without any interpolation used. Unlike some self-driving vehicle datasets where only cars on the road are annotated [20], we also annotate background objects since they can potentially enter the drivable area. Of course, objects that are too small are omitted and our minimum size is 5×15 or 15×5 (based on the aspect ratio of the object). We outsourced the annotation job to Scale AI. In Table B, we compare our annotation with existing datasets: DETRAC [21], KITTI-MOTS [20], MOTS [20], UAVDT [9], Waymo [19], and Youtube-VIS [22].

Table B. Comparison of 2D video object detection datasets. For surveillance camera setups, the cameras are either stationary or have limited motion. For ego-vehicle setups, the scene dynamics evolve quickly, as (1) the ego-vehicle is traveling fast, and (2) other objects are much closer to the camera and thus have a higher speed in the image space. Our contributed dataset (annotation) is a high-frame-rate and high-resolution multi-class one compared to existing datasets

Name	Camera Setup	Image Res	Image FPS	Annot FPS	Classes	Boxes
DETRAC	Survelliance	960×54	30	6	4	1.21M
KITTI-MOTS	Ego-Vehicle	1242×375	10	10	2	46K
MOTS	Generic	1920×1080	30	30	2	30K
UAVDT	UAV Survelliance	1080×540	30	30	1	842K
Waymo	Ego-Vehicle	1920×1280	10	10	4	11.8M
Youtube-VIS	Generic	1280×720	30	6	40	131K
Argoverse-HD (Ours)	Ego-Vehicle	1920×1200	30	30	8	250K

A.5 Experiment Settings

Platforms The CPU used in our experiments is Xeon Gold 5120, and the GPU is Geforce GTX 1080 Ti. The software environment is PyTorch 1.1 with CUDA 10.0.

Timing The setup which we time single-frame algorithms mimics the scenario in real-world applications. The offline pipeline involves several steps: loading data from the disk, image pre-processing, neural network forward pass, and result post-processing. Our timing excludes the first step of loading data from the disk. This step is mainly for dataset-based evaluation. In actual embodied applications, data come from sensors instead of disks. This is implemented by loading the entire video to the main memory before the evaluation starts. In summary, our timing (*e.g.*, the last column of Table 1) starts at the time when

the algorithm receives the image in the main memory, and ends at the time when the results are available in the main memory (instead of in the GPU memory).

A.6 Alternate Task: Instance Segmentation

Table C. Instance segmentation overhead compared with object detection. This table lists runtimes of several methods with and without the mask head, and their differences are the extra cost which one has to pay for instance segmentation. All numbers are milliseconds except the scale column and the last column. The average overhead is 17ms or 13%

Method	Scale	w/o Mask	w/ Mask	Overhead	Overhead
Mask R-CNN ResNet 50	0.2	34.3	41.4	7.1	21%
	0.25	36.1	44.3	8.2	23%
	0.5	56.7	65.6	8.8	16%
	0.75	92.7	101.0	8.3	9%
	1.0	139.6	147.7	8.1	6%
Mask R-CNN ResNet 101	0.2	38.4	46.4	7.9	21%
	0.25	40.9	48.7	7.8	19%
	0.5	68.8	76.4	7.6	11%
	0.75	119.7	127.1	7.5	6%
	1.0	183.8	190.8	7.0	4%
Cascade MRCNN ResNet 50	0.2	60.9	66.0	5.1	8%
	0.25	59.2	69.1	9.9	17%
	0.5	80.0	95.4	15.3	19%
	0.75	118.1	133.8	15.7	13%
	1.0	164.6	181.9	17.3	10%
Cascade MRCNN ResNet 101	0.2	66.4	71.0	4.6	7%
	0.25	65.4	75.2	9.7	15%
	0.5	92.2	106.6	14.4	16%
	0.75	143.4	159.2	15.8	11%
	1.0	208.2	225.1	16.9	8%

In the main text, we propose a meta-benchmark and mention that it can be instantiated with different tasks. In this section, we include *full benchmark evaluation* for streaming instance segmentation.

Instance segmentation is a more fine-grained task than object detection. This creates challenges for streaming evaluation as annotation becomes more expensive and forecasting is not straight-forward. We address these two issues by leveraging pseudo ground truth and warping masks according to the forecasted bounding boxes.

Another issue which we observed is that off-the-shelf pipelines are usually designed for benchmark evaluation or visualization. First, similar to object detection, we adopt GPU image pre-processing by default. Second, we found that

more than 90% of the time within the mask head of Mask R-CNN is spent on transforming masks from the RoI space to the image space and compressing them in a format to be recognized by the COCO evaluation toolkit. Clearly, compression can be disabled for streaming perception. We point out that mask transformation can also be disabled. In practice, masks are used to tell if a specific point or region contains the object. Instead of transforming the mask (which involves object-specific image resizing operations), we can transform the query points or regions, which is simply a linear transformation over points or control points. Therefore, our timing does not include RoI-to-image transformation or mask compression. Furthermore, this also implies that we do not pay an additional cost for masks in forecasting, since only the box coordinates are updated but the masks remain in the RoI space.

For the instance segmentation benchmark, we use the same dataset and the same method HTC [5] for the pseudo ground truth as for detection, and we include 4 methods: Mask R-CNN [12] and Cascade Mask R-CNN [3] with ResNet 50 and ResNet 101 backbones. Since these are hybrid methods that produce both instance boxes and masks, we can measure the overhead of including masks as the difference between runtime with and without the mask head in Table 1. We find that the average overhead is around 13%. We include the streaming evaluation in Tables D and E (with forecasting).

Table D. Streaming evaluation for instance segmentation. We find that *many of our observations for object detection still hold for instance segmentation*: (1) AP drops significantly when moving from offline to real time, (2) the optimal "sweet spot" is not the fastest algorithm but the algorithm with runtime more than the unit frame interval, and (3) both our dynamic scheduling and infinite GPUs further boost the performance. Note that the absolute numbers might appear higher than the tables in the main text since we use pseudo ground truth here

ID	Method	Detector	AP	AP_L	AP_M	AP_S	AP_{50}	AP_{75}	Runtime
1	Accurate (Offline)	Cascade MRCNN R50 @ s1.0	63.1	63.0	60.9	47.9	81.6	69.4	225.1
2	Accurate	Cascade MRCNN R50 @ s1.0	11.8	11.5	8.1	5.4	20.4	11.1	225.1
3	Fast	Mask R-CNN R50 @ s0.2	8.3	16.5	2.1	0.0	13.6	8.3	**41.4**
4	Optimized	Mask R-CNN R50 @ s0.5	**17.2**	**19.9**	**13.8**	**5.2**	**31.8**	**15.1**	65.6
5	+ Scheduling (Alg. 1)	Mask R-CNN R50 @ s0.5	18.3	21.4	14.9	5.8	33.5	16.4	65.5
6	+ Infinite GPUs	Mask R-CNN R50 @ s0.75	20.6	20.0	19.0	9.1	38.4	18.2	100.8

A.7 Alternate Hardware: Tesla V100

In the main text, we propose a meta-benchmark and mention that it can be instantiated with different hardware platforms. In this section, we include *full benchmark evaluation* for streaming detection with Tesla V100 (a faster GPU than GTX 1080 Ti used in the main text).

Table E. Streaming evaluation for instance segmentation with forecasting. Despite that we only forecast boxes and warp masks accordingly, we still observe significant improvement from forecasting for mask AP. The optimized algorithm for row 1 is Mask R-CNN ResNet 50 @ s0.5, and for row 2 is Mask R-CNN ResNet 50 @ s0.75

ID	Method	AP	AP_L	AP_M	AP_S	AP_{50}	AP_{75}
1	Detection + Scheduling + Association + Forecasting	24.1	32.4	23.0	6.0	43.7	22.0
2	+ Infinite GPUs	29.2	30.7	30.2	11.4	53.0	26.7

While our benchmark is hardware dependent, the method of evaluation generalizes across hardware platforms, and our conclusions largely hold when the hardware environment changes. We follow the same setup as in the experiments in the main text, except that we use Tesla V100 from Amazon Web Services (EC2 instance of type `p3.2xlarge`). We provide the results for detection, forecasting, and tracking in Tables F, G, and H, respectively. We see that *the improvement due to better hardware is largely orthogonal to the algorithmic improvement* proposed in the main text.

Table F. Performance of detectors for streaming perception on Tesla V100 (a faster GPU than the Geforce GTX 1080 Ti used in the main text). By comparing with Table 1 in the main text, we see that runtime is shortened and the AP is increased due to the boost of hardware performance. Different from Table 1, we only consider GPU image pre-processing here for simplicity. Interestingly, with additional computation power, Tesla V100 enables more expensive models like input scale 0.75 (row 4) and Cascade Mask R-CNN (row 5) to be the optimal configurations (detector and scale) under their corresponding settings. Note that the improvement from our dynamic scheduler is orthogonal to the boost from hardware performance

ID	Method	Detector	AP	AP_L	AP_M	AP_S	AP_{50}	AP_{75}	Runtime
1	Accurate (Offline)	HTC @ s1.0	38.0	64.3	40.4	17.0	60.5	38.5	338.0
2	Accurate	HTC @ s1.0	8.2	12.3	5.1	1.6	15.3	7.6	338.0
3	Fast	RetinaNet R50 @ s0.25	6.4	17.3	0.6	0.0	11.9	6.0	**43.3**
4	Optimized	Mask R-CNN R50 @ s0.75	13.0	22.2	9.5	**2.3**	**27.6**	10.9	72.1
5	+ Scheduling (Alg. 1)	Cascade MRCNN R50 @ s0.5	**14.0**	**28.8**	**9.9**	1.0	26.8	**12.2**	60.2
6	+ Infinite GPUs	Mask R-CNN R50 @ s1.0	15.9	24.1	13.2	4.9	34.2	13.3	98.8

Table G. Streaming perception with joint detection, association, and forecasting on Tesla V100 (corresponding to Table 2 in the main text). We observe similar boost as in the detection only setting (Table F). The "re-optimize detection" step finds that Mask R-CNN R50 @ s1.0 outperforms Cascade Mask R-CNN R50 @ s0.5 with forecasting (row2), and it also happens to be the optimal detector with infinite GPUs (row 3)

ID	Method	AP	AP_L	AP_M	AP_S	AP_{50}	AP_{75}
1	Detection + Scheduling + Association + Forecasting	18.2	42.7	16.1	1.1	30.9	17.7
2	+ Re-optimize Detection	**19.6**	**33.0**	**19.2**	**5.3**	**38.5**	**17.9**
3	+ Infinite GPUs	22.9	38.7	23.1	6.9	43.8	21.2

Table H. Streaming perception with joint detection, visual tracking, and forecasting on Tesla V100 (corresponding to Table 3 in the main text). We find the similar conclusions that visual tracking with forecasting does not outperform association with forecasting in the single GPU case and achieves comparable performance in the infinite GPU case

ID	Method	AP	AP_L	AP_M	AP_S	AP_{50}	AP_{75}
1	Detection + Visual Tracking	12.6	21.5	9.0	2.2	27.1	10.5
2	+ Forecasting	**18.0**	**34.7**	**16.8**	**3.2**	**36.0**	**16.4**
3	+ Infinite GPUs w/o Forecasting	14.4	24.2	11.2	2.8	30.6	12.0
4	+ Forecasting	**22.8**	**38.6**	**23.0**	**6.9**	**43.7**	**21.0**

B Solution Details

B.1 Dynamic Scheduling

In the main text, we propose the dynamic scheduling algorithm (Alg. 1) to reduce temporal aliasing. Such an algorithm is counter-intuitive in that it minimizes latency by sometimes sitting idle. Here, we provide a theoretical analysis that justifies its superiority over naive idle-free scheduling and we analyze its empirical performance.

We assume no concurrency (*i.e.*, a single job at a time) and that jobs are not interruptible. For notational simplicity, we assume a fixed input frame rate where frame x_i is the frame available at time $i \in \{0, \ldots, T-1\}$ (*i.e.*, zero-based indexing), and therefore i can be used to denote both frame index and time. *We assume time (time axis, runtime, and latency) is represented in the units of the number of frames.* We also assume g to be a *single-frame* algorithm, and the streaming algorithm f is thus composed of g and the scheduling policy. No tracking or forecasting is used in the discussion below. Denote the runtime of g as r. Let k_j be the time index of the single-frame input that was processed to generate output $o_j = (\hat{y}_j, s_j)$: if $\hat{y}_j = g(x_i)$, then $k_j = i$.

Definition (Temporal Mismatch) When the benchmark queries for the state of the world at frame i, the temporal mismatch is $\delta_i := i - k_j$, where $j = \arg\max_{j'} s_{j'} < i$. If there is no output available, $\delta_i := 0$. Denote the average temporal mismatch over the entire sequence as $\bar{\delta}$.

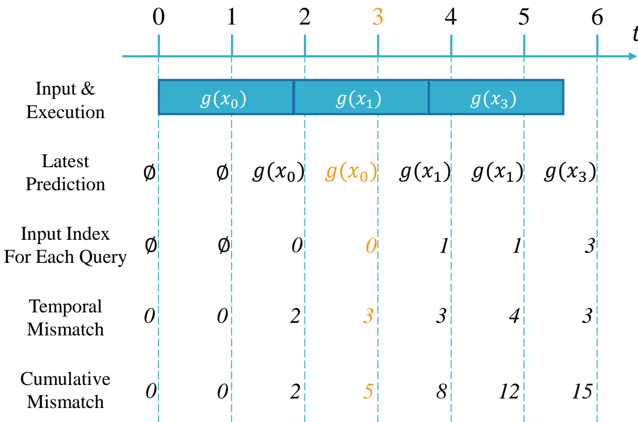

Fig. B. Temporal mismatch for single-frame algorithms. Take $t = 3$ (query index $i = 3$) as an example (highlighted in orange): when the benchmark queries for y_3, the latest prediction is $g(x_0)$, whose input index is 0, thus leading to a temporal mismatch of 3 (frames).

Intuitively, the temporal mismatch measures the latency of a streaming algorithm f in the unit of the number of frames (Figure B). This latency is typically

higher than the runtime of the single-frame algorithm g itself due to the blocking effect of consecutive execution blocks. For example, in Figure B, although runtime $r < 2$, the average mismatch $\bar{\delta} = 15/7 > 2$ for $T = 7$. We will show that this mismatch can actually be minimized through proper scheduling. Note that if an algorithm g always has runtime smaller than the duration of one frame, no schedule can reduce the mismatch. Therefore we assume that $r > 1$ (*runtime r of algorithm g is greater than 1*) for the reasoning below.

Definition (Scheduling Policy) A *scheduling policy*, or *policy* for short, is a Boolean-valued function $(t, r) \to \{\text{true}, \text{false}\}$ where t is the finishing time of an execution block and r is runtime of algorithm g. The function is only called when algorithm g finishes processing a frame, and returns true if g should wait for the next frame and false otherwise (whereby g should process the latest available frame).

Under the above definition, idle-free is a policy that always returns false and it is the commonly adopted approach for asynchronous processing. However, we will show shortly that it is actually suboptimal. Note that this definition directly rules out trivial suboptimal cases, such as waiting beyond the next available frame. For simplicity, we assume that *runtime r is constant*, and that all policies start processing the first frame immediately at $t = 0$. This avoids the degenerate case where an algorithm processes nothing and yields a zero cumulative temporal mismatch.

Definition (Optimality) An *optimal scheduling policy* is the one that minimizes the cumulative temporal mismatch, or equivalently $\bar{\delta}$, for given runtime r and sequence length T with $T \gg r$.

The assumption of $T \gg r$ avoids discussion about the hypothetical cases when T is small (*e.g.*, a sequence with only a few frames)

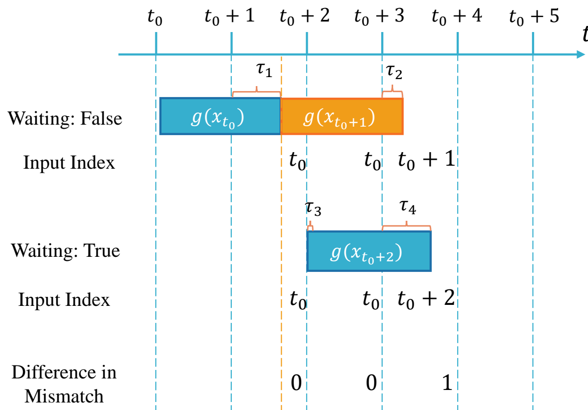

Fig. C. A shrinking-tail execution block (orange) increases temporal mismatch.

With the problem well-defined, we now show that our shrinking-tail policy (Alg. 1) outperforms naive idle-free policy. The reasoning is based on the important concept of *tail*: $\tau(t) := t - \lfloor t \rfloor$. We denote $\tau(r)$ as τ_r for short. Note that the integral part of the runtime does not contribute to the temporal quantization effect and thus we focus on the discussion of $1 < r \leq 2$ (since r is assumed to be greater than 1). Additionally, we split our discussion into 3 different cases: $1 < r < 1.5$, $1.5 \leq r < 2$ and $r = 2$ (or any positive integers).

Case 1 We start by the case with $1 < r < 1.5$ (*i.e.*, $\tau_r < 0.5$). We first observe a special type of execution block:

Definition (Shrinking-Tail Block) Denoting the start and end time of an execution block as t_1 and t_2, a *shrinking-tail block* is an execution block such that $\tau(t_1) > \tau(t_2)$.

As shown in Figure C, a shrinking-tail block increases temporal mismatch.

Definition (Shrinking-Tail Cycle) A sequence of execution blocks can be divided into segments using shrinking-tail block or idle gap as dividers. A *shrinking-tail cycle* is a set of queries covered by the segment between these dividers. Specially, the cycle starts from the 0-th query, the last query of a shrinking-tail block, or the query at the end of the idle gap, as shown in Figure D. The cycle ends either when the sequence ends or the next cycle starts. The length of a cycle is the number of queries it has covered.

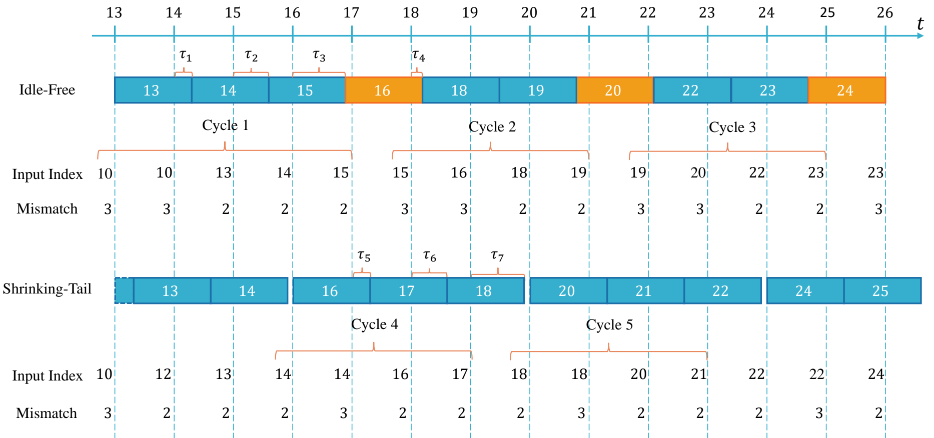

Fig. D. Shrinking-tail cycle (for $1 < r < 1.5$). Intuitively, blocks within each shrinking-tail cycle has tails increasing ($\tau_1 < \tau_2 < \tau_3$ and $\tau_5 < \tau_6 < \tau_7$). It ends when tail decreases or there is an idle gap, and thus the tail "shrinks".

We now analyze the average mismatch $\bar{\delta}^{(c)}$ within a shrinking-tail cycle for the shrinking-tail policy ($\bar{\delta}^{(c)}_{\text{st}}$) and the idle-free policy ($\bar{\delta}^{(c)}_{\text{if}}$) respectively.

As shown in Figure D, for the shrinking-tail policy, a unique type of shrinking-tail cycle occurs periodically (*e.g.*, Cycle 4 covering queries from 16 to 19). Its

cycle length is $c+1$, where c satisfies that $c\tau_r < 1$ and $(c+1)\tau_r \geq 1$. We then have $\bar{\delta}_{\text{st}}^{(c)} = 1/(c+1)+2$. For example, for Cycle 4, all the temporal mismatches are 2 except a single 3 at $i = 17$. So the average mismatch $\bar{\delta}_{\text{st}}$ over the entire sequence will satisfy $\bar{\delta}_{\text{st}} = \bar{\delta}_{\text{st}}^{(c)}$.

For the idle-free policy, there are two types of shrinking-tail cycles (*e.g.*, Cycle 1 and Cycle 2 in Figure D). One of the cycles (*e.g.*, Cycle 1 covering queries from 13 to 17) has length $c+2$ and $\bar{\delta}_{\text{if}}^{(c1)} = 2/(c+2)+2$. The other cycle (*e.g.*, Cycle 2 covering queries from 18 to 21) has length $c+1$ and $\bar{\delta}_{\text{if}}^{(c2)} = 2/(c+1)+2$. For example, for either Cycle 1 or Cycle 2, all the temporal mismatches are 2 except two 3 at $i = 13$ and $i = 14$ (or $i = 18$ and $i = 19$). So the average mismatch $\bar{\delta}_{\text{if}}$ over the entire sequence will satisfy $\bar{\delta}_{\text{if}}^{(c1)} \leq \bar{\delta}_{\text{if}} \leq \bar{\delta}_{\text{if}}^{(c2)}$.

Comparing the two policies, it is easy to see that $\bar{\delta}_{\text{st}}^{(c)} < \bar{\delta}_{\text{if}}^{(c1)}$, and thus we have $\bar{\delta}_{\text{st}} < \bar{\delta}_{\text{if}}$.

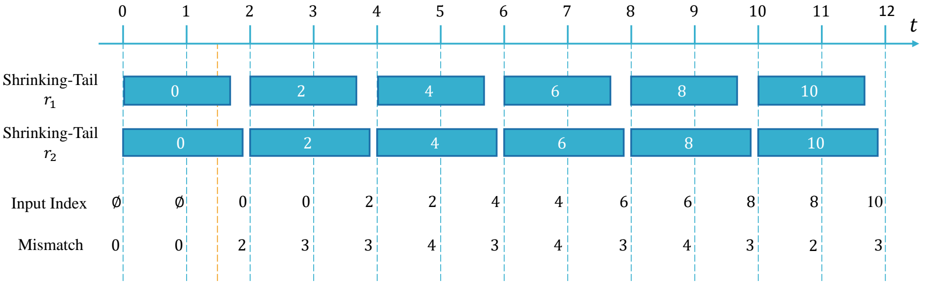

Fig. E. Mismatch is the same for the shrinking-tail policy with different runtime r_1 and r_2 as long as $\lfloor r_1 \rfloor = \lfloor r_2 \rfloor$, $\tau(r_1) \geq 0.5$, and $\tau(r_2) \geq 0.5$.

Case 2 Now we inspect the case with $1.5 \leq r < 2$. Since $\tau(2r) < 0.5 < t_r$, the shrinking-tail policy will output true (waiting) after processing the first frame. The waiting aligns the execution again with the integral time step and thus for the subsequent processing blocks, it also output true (Waiting). In summary, shrinking-tail always outputs true in this case and its pattern in mismatch is agnostic to the specific runtime r (Fig. E). Let $\bar{\delta}_{\text{st}}^{r}$ denote the average temporal mismatch $\bar{\delta}$ for the shrinking-tail policy with runtime r, then we can draw the conclusion that $\bar{\delta}_{\text{st}}^{r_1} = \bar{\delta}_{\text{st}}^{r_2}$ for $\lfloor r_1 \rfloor = \lfloor r_2 \rfloor$, $\tau(r_1) \geq 0.5$, and $\tau(r_2) \geq 0.5$.

We then focus on a particular case that $r = 1.5$. As shown in Figure F, idle-free repeats itself in a period of 3 frames and shrinking-tail repeats itself in a period of 2 frames. Together, they form a joint pattern that repeats itself in a period of 6 frames (their least common multiple). The diagram shows that within each common period, the difference of cumulative mismatch between idle-free and shrinking-tail is increased by 1. And it is the same for all common periods. Therefore, we conclude that $\bar{\delta}_{\text{st}}^{0.5} < \bar{\delta}_{\text{if}}^{0.5}$.

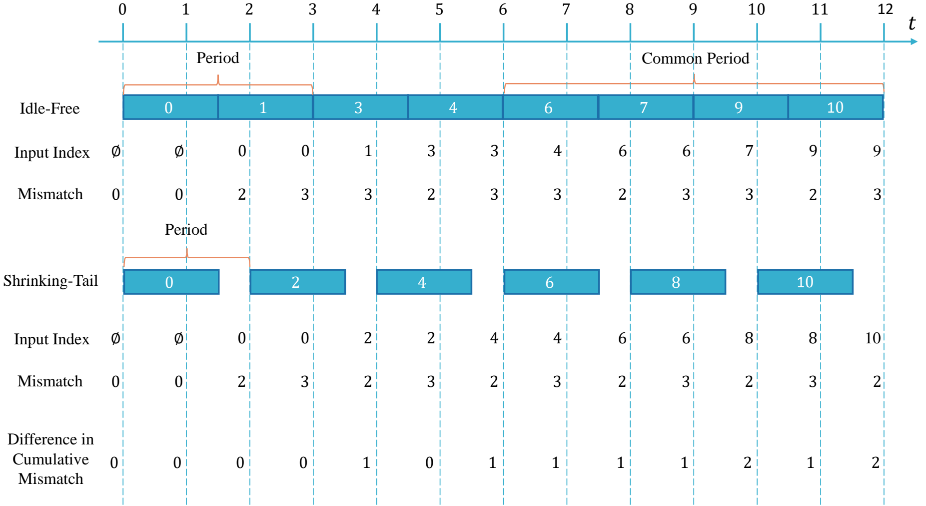

Fig. F. For $r = 1.5$, shrinking-tail achieves less cumulative mismatch than idle-free. Note that each policy has its own repeating period and shrinking-tail always achieves 1 less cumulative mismatch within each common period.

In addition, it is straightforward to see that the cumulative mismatch will not decrease if one increases the runtime r of g: $\bar{\delta}^{r_1} \leq \bar{\delta}^{r_2}$ if $r_1 < r_2$. Therefore, for $1.5 \leq r < 2$,

$$\bar{\delta}^{r}_{\text{st}} = \bar{\delta}^{0.5}_{\text{st}} < \bar{\delta}^{0.5}_{\text{if}} \leq \bar{\delta}^{r}_{\text{if}} \tag{1}$$

Case 3 The last case is a special case of $\tau_r = 0$. It can be easily verified that idle-free is equivalent to shrinking-tail, and thus $\bar{\delta}_{\text{st}} \leq \bar{\delta}_{\text{if}}$.

Summary of the theoretical reasoning Considering all 3 cases, we can draw the conclusion that $\bar{\delta}_{\text{st}} \leq \bar{\delta}_{\text{if}}$, and they equal only when $\tau_r = 0$. By achieving less average mismatch, shrinking-tail outperforms idle-free regardless ($r > 1$).

Practical Performance of Dynamic Scheduling We apply our dynamic schedule (Alg. 1) to a wide suite of detectors under the same settings as our main experiments and summarize the results in Table I. We see that the improvement generalizes to most of the cases. Note that contrary to our theoretical analysis above, in practice runtime is stochastic due to complicated software and hardware scheduling or the model is adaptive to the input.

B.2 Additional Details for Forecasting

We use an asynchronous Kalman filter for our forecasting module. The state representation which we choose is $[x, y, w, h, \dot{x}, \dot{y}, \dot{w}, \dot{h}]$, where $[x, y, w, h]$ are the

Table I. Empirical performance comparison before and after using Alg. 1. We see that our shrinking-tail policy consistently boosts the streaming performance for different detectors and for different input scales. We also observe some failure cases (last two rows), where runtime is close to one frame duration. This is because our theoretical analysis assumes constant runtime, while it is dynamic in practice. Hence, the variance in runtime when it is a boundary value can make a noticeable difference on the performance

Method	AP (Before)	AP (After)	Runtime (ms)	Runtime (frames)
SSD @ s0.5	9.7	9.7	66.7	2.0
RetinaNet R50 @ s0.5	10.9	11.6	54.5	1.6
RetinaNet R101 @ s0.5	9.9	9.9	66.8	2.0
Mask R-CNN R101 @ s0.5	11.0	11.1	68.8	2.1
Cascade MRCNN R50 @ s0.5	11.3	11.7	80.0	2.4
Cascade MRCNN R101 @ s0.5	10.3	11.1	92.2	2.8
HTC @ s0.5	7.9	8.0	240.8	7.2
Mask R-CNN R50 @ s0.25	7.7	7.8	36.1	1.1
Mask R-CNN R50 @ s0.5	12.0	13.0	56.7	1.7
Mask R-CNN R50 @ s0.75	11.5	12.6	92.7	2.8
Mask R-CNN R50 @ s1.0	10.6	10.7	139.6	4.2
RetinaNet R50 @ s0.25	6.9	6.8	33.4	1.0
Mask R-CNN R50 @ s0.2	6.5	6.3	34.3	1.0

top-left coordinates, and width and height of the bounding box, and the remaining four are their derivatives. The state transition is assumed to be linear. We also test with the representation used in SORT [2], which assumes that the area (the product of the width and the height) varies linearly instead of that each of the width and the height varies linearly. We find that such a representation produces lower numbers in AP.

As explained in the main text, Kalman filter needs to be asynchronous and time-varying for streaming perception. Let Δt_k be the time-varying intervals between updates or prediction steps, we pick the transition matrix to be:

$$\mathbf{F}_k = \begin{bmatrix} \mathbf{I}_{4\times 4} & \Delta t_k \mathbf{I}_{4\times 4} \\ & \mathbf{I}_{4\times 4} \end{bmatrix} \tag{2}$$

and the process noise to be

$$\mathbf{Q}_k = \Delta t_k^2 \mathbf{I}_{8\times 8} \tag{3}$$

Intuitively, the process noise is larger the longer between the updates.

All forecasting modules are implemented on the CPU and thus can be parallelized while the detector runs on the GPU. Our batched (over multiple objects) implementation of the asynchronous Kalman filter takes 0.98 ± 0.39ms for the update step and 0.22 ± 0.07ms for the prediction step, which are relatively very small overheads compared to detector runtimes. For scalable evaluation, we assume zero runtime for the association and forecasting module, and implement

forecasting as post-processing of the detection outputs. One might wonder that a simulated post-processing run and an actual real-time parallel execution might have different final APs. We have also implemented the latter for verification purposes. For most settings the differences are within 1%. Although for some settings the difference can reach 3%, we find such fluctuation does not affect the relative rankings.

B.3 Additional Details for Visual Tracking

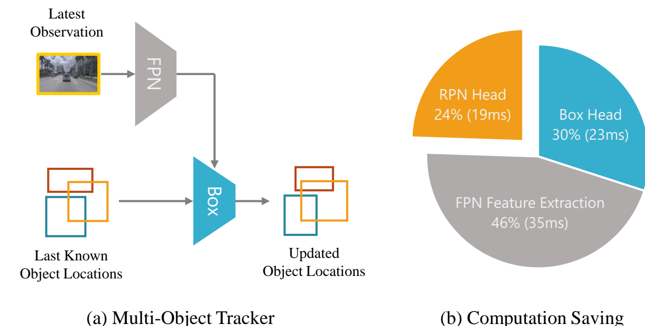

Fig. G. Multi-object visual tracker. The advantage of a visual tracker is that it runs faster than a detector and thus yields lower latency for streaming perception. The multi-object tracker used here is modified from [1]. It is mostly the same as a two-stage detector, except that its box head uses the last known object location as input in place of region proposals. Therefore, we get a computation saving by not running the RPN head. Runtime is measured for Mask R-CNN (ResNet 50) with input scale 0.5.

For our tracking experiments (Section 4.4 in the main text), we adapt and modify the state-of-the-art multi-object tracker [1]. A component breakdown in Fig. G explains how this tracker works and why it has the potential to achieve better performance under the streaming setting.

B.4 Evaluation of Our Meta-Detector *Streamer*

Streamer is introduced in Section 4.3 in the main text. Given a detector and an input scale, Streamer automatically schedules the detector and employs forecasting to compensate for some of its latency. In the single GPU case, our dynamic schedule (Alg. 1) is used and in the infinite GPU case, idle-free scheduling (Fig. 4c) is used. Proper scheduling requires the knowledge of runtime of the algorithm, which is known in the case of benchmark evaluation. When applied in the wild, we can optionally track runtime of the algorithm on unseen data and adjust the scheduling accordingly. The forecasting module is implemented with asynchronous Kalman filter (Section B.2).

Streamer has several key features. First, it enables synchronous processing for an asynchronous problem. Under the commonly studied settings (both offline and

Table J. Performance boost after applying Streamer. "(B)" standards for "Before", and "(A)" standards for "After". The evaluation setting is the same as Table 1 in the main text. This table assumes a *single* GPU, and an infinite GPU counterpart can be found in Table K. Under this setting, we observe significant improvement in AP, ranging from 5% to 78%, and averaging at 34%

Method	Scale	AP(B)	AP(A)	Boost	AP_L(B)	AP_L(A)	Boost
SSD	0.2	9.5	10.4	9%	23.5	28.6	21%
	0.25	9.3	10.6	14%	23.9	31.5	32%
	0.5	9.7	13.5	40%	20.0	32.4	62%
	0.75	6.0	10.7	78%	11.5	19.8	72%
	1.0	4.2	7.3	76%	7.3	12.5	72%
RetinaNet R50	0.2	6.0	6.3	5%	18.1	21.3	17%
	0.25	6.9	7.5	9%	19.8	26.2	33%
	0.5	10.9	14.2	30%	24.1	38.3	59%
	0.75	10.8	16.1	50%	20.2	32.9	63%
	1.0	9.9	14.1	42%	16.7	24.7	48%
RetinaNet R101	0.2	5.4	5.9	9%	14.7	19.8	35%
	0.25	6.5	7.4	14%	18.2	25.8	42%
	0.5	9.9	13.0	31%	21.5	33.6	56%
	0.75	9.9	14.5	47%	18.1	27.7	53%
	1.0	8.9	12.7	42%	14.7	22.0	50%
Mask R-CNN R50	0.2	6.5	7.2	11%	18.0	25.1	40%
	0.25	7.7	9.1	19%	20.1	29.9	49%
	0.5	12.0	16.7	39%	24.3	39.9	64%
	0.75	11.5	17.8	54%	19.5	33.3	71%
	1.0	10.6	15.0	42%	16.6	25.0	50%
Mask R-CNN R101	0.2	6.3	7.2	14%	16.7	24.1	45%
	0.25	7.6	9.0	17%	19.3	28.5	48%
	0.5	11.0	15.2	39%	21.6	35.4	64%
	0.75	10.0	15.3	52%	16.8	28.0	67%
	1.0	8.8	12.4	42%	13.7	21.2	55%
Cascade MRCNN R50	0.2	6.2	7.8	25%	15.4	25.5	66%
	0.25	7.5	9.6	28%	18.4	30.1	63%
	0.5	11.3	16.4	45%	22.6	37.5	66%
	0.75	10.9	16.7	54%	18.6	29.8	60%
	1.0	10.1	15.7	55%	15.4	25.3	64%
Cascade MRCNN R101	0.2	6.1	7.3	20%	15.2	23.1	52%
	0.25	7.4	9.5	28%	17.6	29.6	69%
	0.5	10.3	15.4	49%	20.5	34.1	66%
	0.75	9.5	14.7	54%	16.1	26.1	62%
	1.0	8.8	12.9	46%	13.7	21.8	59%
HTC	0.2	5.6	6.8	22%	12.0	17.0	42%
	0.25	6.3	8.3	31%	13.0	19.8	53%
	0.5	7.9	10.8	38%	13.3	19.9	49%
	0.75	7.1	8.6	22%	11.4	14.8	30%
	1.0	6.4	7.2	12%	9.6	11.4	18%

Table K. Performance boost after applying Streamer. "(B)" standards for "Before", and "(A)" standards for "After". The evaluation setting is the same as Table 1 in the main text. This table assumes *infinite* GPUs, and a single GPU counterpart can be found in Table J. Under this setting, we observe significant improvement in AP, ranging from 4% to 80%, and averaging at 32%

Method	Scale	AP(B)	AP(A)	Boost	AP_L(B)	AP_L(A)	Boost
SSD	0.2	9.9	10.6	7%	25.5	29.4	15%
	0.25	9.6	10.7	12%	24.9	31.7	27%
	0.5	11.3	14.7	30%	24.1	35.4	47%
	0.75	8.0	13.3	66%	14.6	25.6	76%
	1.0	5.5	9.8	80%	10.0	16.5	65%
RetinaNet R50	0.2	6.1	6.3	4%	18.6	21.3	15%
	0.25	7.1	7.6	8%	21.4	27.1	26%
	0.5	12.3	14.7	20%	28.1	40.1	42%
	0.75	13.1	18.0	37%	24.3	37.8	56%
	1.0	11.7	17.3	48%	19.5	31.3	60%
RetinaNet R101	0.2	5.5	6.0	9%	15.3	20.1	32%
	0.25	6.7	7.5	12%	18.8	26.1	38%
	0.5	11.3	14.0	24%	25.3	38.1	50%
	0.75	11.8	17.0	44%	21.3	34.3	61%
	1.0	10.8	16.3	51%	18.2	28.2	55%
Mask R-CNN R50	0.2	6.7	7.4	10%	20.0	26.2	31%
	0.25	7.8	9.2	17%	20.8	30.1	45%
	0.5	13.9	17.4	26%	29.0	42.6	47%
	0.75	14.4	20.3	40%	24.3	38.5	59%
	1.0	12.4	18.7	51%	19.4	31.4	62%
Mask R-CNN R101	0.2	6.5	7.3	13%	17.4	24.3	40%
	0.25	7.9	9.1	15%	20.5	28.9	41%
	0.5	11.9	16.2	36%	23.7	38.4	62%
	0.75	12.4	18.5	49%	20.3	35.3	74%
	1.0	10.6	16.2	53%	16.9	27.7	64%
Cascade MRCNN R50	0.2	7.0	7.9	13%	18.9	26.5	40%
	0.25	8.5	9.9	16%	22.3	31.7	42%
	0.5	12.9	17.6	37%	26.0	41.2	58%
	0.75	13.2	19.9	51%	22.1	36.5	65%
	1.0	12.6	19.8	57%	19.0	31.8	67%
Cascade MRCNN R101	0.2	6.8	7.9	17%	17.8	26.6	49%
	0.25	8.3	9.8	18%	21.0	31.7	50%
	0.5	12.6	17.0	35%	25.0	38.5	54%
	0.75	11.4	17.7	56%	19.0	32.7	72%
	1.0	10.5	16.6	59%	16.7	27.6	65%
HTC	0.2	6.3	8.0	27%	14.0	21.8	55%
	0.25	7.3	9.8	34%	15.7	25.5	62%
	0.5	9.2	13.7	50%	16.3	26.9	65%
	0.75	8.2	11.4	39%	13.2	20.5	55%
	1.0	7.4	9.3	25%	11.1	15.8	43%

online), computation is synchronous in that the outputs and the inputs have a natural one-to-one correspondence. Therefore, many existing temporal reasoning models assume that the inputs are at a uniform rate and each input corresponds to an output [8,11,10]. In the real-time setting, however, such a relationship does not exist due to the latency of the algorithm, *i.e.*, the number of outputs can be arbitrary. Streamer converts detectors with arbitrary runtimes into systems that output at a designated fixed rate. In short, it abstracts away the asynchronous nature of the problem and therefore allows downstream synchronous processing. Second, by adopting forecasting, Streamer significantly boosts the performance of streaming perception. In Tables J and K, we evaluate the detection AP before and after applying our meta-detector. We observe relative improvement from 4% to 80% with an average of 33% in detection AP under 80 different settings (8 detectors $\times$ 5 image scales $\times$ 2 compute models). Note that the difference of this evaluation and benchmark evaluation in the main text is that we fix the detector and input scale here, while benchmark evaluation searches over the best configuration of detectors and input scales. For the infinite GPU settings, we discount the boost from additional compute itself.

B.5 Implementation Details

Detectors We experiment with a large suite of object detectors: SSD [16], RetinaNet [14], Mask R-CNN [12], Cascade Mask R-CNN [3], and HTC [5]. The "optimized" and "re-optimized" rows in all tables represent the optimal configuration over all detectors and all input scales of 0.2, 0.25, 0.5, 0.75, and 1.0. We adopt mmdetection codebase [6] (one of the fastest open-source implementation for Mask R-CNN) for object detectors. Note that for all detectors, the implementation has reproduced both the accuracy and runtime reported in the original papers.

Potentially better implementation We acknowledge that there are additional bells and whistles to reduce runtime of object detectors, which might further improve the results on our benchmark. We focus on general techniques instead of device- or application-specific ones. For example, we have explored GPU image pre-processing, which is applicable to all GPUs. Another implementation technique is to use half-precision floating-point numbers (FP16), which we have not explored, since it will only pay off for certain GPUs that have been optimized for FP16 computation (it is reported that FP16 yields only marginal testing time speed boost on 1080 Ti [7]).

C Additional Baselines

C.1 Forecasting Baselines

We have also tested linear extrapolation (*i.e.*, constant velocity) and quadratic extrapolation for forecasting detection results. We include an illustration of linear

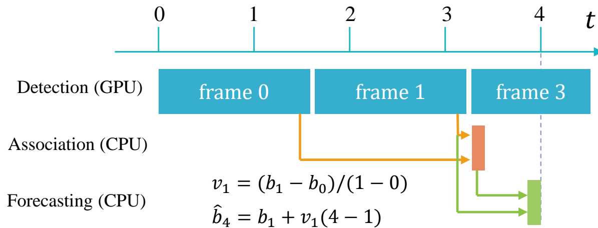

Fig. H. Scheduling for linear forecasting. The scheduling is similar as with the Kalman filter case in that both are asynchronous. The difference is that linear forecasting does not explicitly maintain a state representation but only stores two latest detection results. Association takes place immediately after a new detection result becomes available, and it links the bounding boxes in two consecutive detection results and computes a velocity estimate. Forecasting takes place right before the next time step, and it uses linear extrapolation to produce an output as the estimation of the current world state. The equations represent the computation for reporting to benchmark query at $t = 4$. b is a simplified representation for object location. At this time, only detection results for frame 0 and 1 are available, but through association and forecasting, the algorithm can make a better prediction for the current world state.

Table L. Comparison of different forecasting methods for streaming perception. We see that both linear and Kalman filter forecasting methods significantly improve the streaming performance. Kalman filter further outperforms the linear forecasting. The quadratic forecasting decreases the AP, suggesting that high-order extrapolation is not suitable for this task. The detection used here is Mask R-CNN ResNet 50 @ s0.5 with dynamic scheduling (Alg. 1)

ID	Method	AP	AP_L	AP_M	AP_S	AP_{50}	AP_{75}
1	No Forecasting	13.0	26.6	9.2	1.1	26.8	11.1
2	Linear (constant velocity)	15.7	38.1	13.8	1.1	30.2	14.8
3	Quadratic	9.7	23.8	6.6	0.4	21.4	7.9
4	Kalman filter	**16.7**	**39.8**	**14.9**	**1.2**	**31.2**	**16.0**

forecasting in Fig. H, and the quadratic counterpart is a straight-forward extension that involves three latest detection results. Though they produce inferior results than Kalman filter, we include the results in Table L for completeness.

C.2 An End-to-End Baseline

In the main text, we break down the streaming detection task into detection, tracking, and forecasting for modular analysis. Alternatively, it is also possible to train a model that directly outputs detection results in the future. F2F [17] is one such model. Building upon Mask R-CNN, it does temporal reasoning and forecasting at the level of FPN feature maps. Note that no explicit tracking is performed. In this section, we compare against this end-to-end baseline in both offline and streaming settings.

In the offline setting, the algorithm is given s frames as input history, and outputs detection results for t frames ahead. This is the same as the evaluation in [17]. We set both s and t to be 3, as the optimal detector in our forecasting experiments (Table 2) has runtime of 2.78 frames. Since F2F forecasts at the FPN feature level, it is agnostic to second stage tasks. In our evaluation, we focus on the bounding box detection task instead of instance segmentation. Also, we conduct experiments on Argoverse-HD, consistent with the setting in our other experiments. Due to a lack of annotation, we adopt pseudo ground truth (Section A.2) for training (data from the original training set of Argoverse 1.1 [4]). We implement our own version of F2F based on mmdetection (instead of Detectron as done in [17]). We train the model for 12 epochs end-to-end (a 50% longer schedule than combined stages in [17]). For a fair comparison, we also finetuned the detectors on Argoverse with the same pseudo ground truth. For Mask R-CNN ResNet 50 at scale 0.5, it boosts the offline box AP from 19.4 to 22.9. We use this finetuned detector in our method to compare against F2F. The results are summarized in Table M. We see that an end-to-end solution does not immediately boost the performance. We believe that it is still an open problem on how to effectively replace tracking and forecasting with an end-to-end solution.

In the streaming setting, F2F can be viewed as a detector that compensates its own latency. The results are summarized in Table N. We see that F2F is too expensive compared with other streaming solutions, showing that forecasting

Table M. Standard offline forecasting evaluation for the end-to-end method F2F [17]. The goal is to forecast 3 frames into the future. Surprisingly, the more expensive F2F method performs worse than the simpler Kalman filter in terms of the overall AP

ID	Method	AP	AP_L	AP_M	AP_S	AP_{50}	AP_{75}
1	None (copy last)	13.4	24.3	10.9	1.9	27.9	11.3
2	Linear	16.3	34.8	16.8	1.8	32.9	14.3
3	Kalman filter	**19.1**	40.3	19.8	**2.6**	**35.8**	**17.7**
4	F2F	18.3	**41.0**	**20.0**	2.5	33.9	17.1

can help only if it is fast under our evaluation. Note that the detectors (row 1–2) are not finetuned as in the offline case, which means that they can be further improved.

Table N. Streaming evaluation for the end-to-end method F2F [17]. The setting is the same as the experiments in the main text. Rows 1 and 2 are the optimized detector and the Kalman filter forecasting solution from the main text. The underlying detectors used are Mask R-CNN ResNet 50 at scale 0.5 and scale 0.75 respectively. Row 3 suggests that F2F has a low streaming AP, due to its forecasting module being very expensive (last column, runtime in milliseconds). For diagnostics purpose, we assume F2F to run as fast as our optimized detector (row 4), and arm it with our scheduling algorithm (row 5). But even so, F2F still under-performs the simple Kalman filter solution

ID	Method	AP	AP_L	AP_M	AP_S	AP_{50}	AP_{75}	Runtime
1	Detection	12.0	24.3	7.9	1.0	25.1	10.1	**56.7**
2	+ Scheduling (Alg. 1) + KF	**17.8**	**33.3**	**16.3**	**3.2**	**35.2**	**16.5**	92.7
3	F2F	6.2	11.1	3.4	0.8	13.1	5.2	321.6
4	F2F (Simulated Fast)	14.1	29.1	12.7	1.9	28.9	12.0	92.7
5	+ Scheduling (Alg. 1)	15.6	33.0	15.2	2.1	30.7	13.9	92.7

References

1. Bergmann, P., Meinhardt, T., Leal-Taixé, L.: Tracking without bells and whistles. In: ICCV (2019) 16
2. Bewley, A., Ge, Z., Ott, L., Ramos, F., Upcroft, B.: Simple online and realtime tracking. In: ICIP (2016) 15
3. Cai, Z., Vasconcelos, N.: Cascade R-CNN: Delving into high quality object detection. In: CVPR (2018) 7, 19
4. Chang, M.F., Lambert, J.W., Sangkloy, P., Singh, J., Bak, S., Hartnett, A., Wang, D., Carr, P., Lucey, S., Ramanan, D., Hays, J.: Argoverse: 3D tracking and forecasting with rich maps. In: CVPR (2019) 3, 5, 21
5. Chen, K., Pang, J., Wang, J., Xiong, Y., Li, X., Sun, S., Feng, W., Liu, Z., Shi, J., Ouyang, W., Loy, C.C., Lin, D.: Hybrid task cascade for instance segmentation. In: CVPR (2019) 2, 7, 19
6. Chen, K., Wang, J., Pang, J., Cao, Y., Xiong, Y., Li, X., Sun, S., Feng, W., Liu, Z., Xu, J., Zhang, Z., Cheng, D., Zhu, C., Cheng, T., Zhao, Q., Li, B., Lu, X., Zhu, R., Wu, Y., Dai, J., Wang, J., Shi, J., Ouyang, W., Loy, C.C., Lin, D.: MMDetection: Open mmlab detection toolbox and benchmark. arXiv preprint arXiv:1906.07155 (2019) 19
7. Chen, Y., , Han, C., Li, Y., Huang, Z., Jiang, Y., Wang, N., Zhang, Z.: SimpleDet: A simple and versatile distributed framework for object detection and instance recognition. JMLR (2019) 19
8. Donahue, J., Hendricks, L.A., Rohrbach, M., Venugopalan, S., Guadarrama, S., Saenko, K., Darrell, T.: Long-term recurrent convolutional networks for visual recognition and description. In: CVPR (2015) 19
9. Du, D., Qi, Y., Yu, H., Yang, Y.F., Duan, K., Li, G., Zhang, W., Huang, Q., Tian, Q.: The unmanned aerial vehicle benchmark: Object detection and tracking. In: ECCV (2018) 5
10. Feichtenhofer, C., Fan, H., Malik, J., He, K.: Slowfast networks for video recognition. In: ICCV (2019) 19
11. Girdhar, R., Carreira, J., Doersch, C., Zisserman, A.: A better baseline for AVA. ArXiv abs/1807.10066 (2018) 19
12. He, K., Gkioxari, G., Dollár, P., Girshick, R.B.: Mask R-CNN. In: ICCV (2017) 4, 7, 19
13. Li, M., Yumer, E., Ramanan, D.: Budgeted training: Rethinking deep neural network training under resource constraints. In: ICLR (2020) 4
14. Lin, T.Y., Goyal, P., Girshick, R., He, K., Dollár, P.: Focal loss for dense object detection. In: ICCV (2017) 3, 19
15. Lin, T.Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., Zitnick, C.L.: Microsoft COCO: Common objects in context. In: ECCV (2014) 4, 5
16. Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S.E., Fu, C.Y., Berg, A.C.: SSD: Single shot multibox detector. In: ECCV (2016) 3, 19
17. Luc, P., Couprie, C., LeCun, Y., Verbeek, J.: Predicting future instance segmentations by forecasting convolutional features. In: ECCV (2018) 21, 22
18. Redmon, J., Divvala, S., Girshick, R., Farhadi, A.: You only look once: Unified, real-time object detection. In: CVPR (2016) 3
19. Sun, P., Kretzschmar, H., Dotiwalla, X., Chouard, A., Patnaik, V., Tsui, P., Guo, J., Zhou, Y., Chai, Y., Caine, B., Vasudevan, V., Han, W., Ngiam, J., Zhao, H., Timofeev, A., Ettinger, S., Krivokon, M., Gao, A., Joshi, A., Zhang, Y., Shlens, J.,

Chen, Z., Anguelov, D.: Scalability in perception for autonomous driving: Waymo open dataset (2019) 5
20. Voigtlaender, P., Krause, M., Ošep, A., Luiten, J., Sekar, B.B.G., Geiger, A., Leibe, B.: MOTS: Multi-object tracking and segmentation. In: CVPR (2019) 5
21. Wen, L., Du, D., Cai, Z., Lei, Z., Chang, M., Qi, H., Lim, J., Yang, M., Lyu, S.: DETRAC: A new benchmark and protocol for multi-object detection and tracking. arXiv abs/1511.04136 (2015) 5
22. Yang, L., Fan, Y., Xu, N.: Video instance segmentation. In: ICCV (2019) 5