Adaptive object detection algorithms for resource constrained autonomous robotic systems

Joe Pappas[a], Venkat R. Dasari[b], Billy E. Geerhart[b], David M. Alexander[b], Peng Wang[b], and Somali Chaterji[a]

[a]Purdue University, West Lafayette, IN 47907
[b]DEVCOM Army Research Laboratory, Aberdeen Proving Ground, United States

ABSTRACT

We optimized and deployed the adaptive framework Virtuoso that can maintain real-time object detection even when experiencing high contention scenarios. The original Virtuoso framework uses an adaptive algorithm for the detection frame followed by a low cost algorithm for the tracker frame which uses down-sampled images to reduce computation. One of our optimizations include detaching the single synchronous thread for detection and tracking into two parallel threads. This multi-threaded implementation allows for computationally high cost detection algorithms to be used while still maintaining real-time output from the tracker thread. Another optimization we developed uses multiple down-sampled images to initialize each tracker based on the size of the input box; the multiple down-sampled images allow each tracker to choose the optimal image size for the box that it is tracking rather than a single down-sampled image being used for all trackers.

Keywords: object detection, virtuoso, adaptive, optimization

1. INTRODUCTION

Our research focuses on optimizing and deploying autonomous robotic algorithms to increase a robots functionality and reliability. Optimized algorithms are particularly important for robots operating under resource constraints while under high contention; in such a scenario the ability for each algorithm to maintain real-time processing is essential for navigation in dynamic environments. Previously we have focused exclusively on optimizing an algorithm by reducing the compute cost such that the algorithm would perform well under all contention scenarios, but for this work we are enhancing the adaptive object detection framework Virtuoso.

The Virtuoso framework attempts to balance the trade-off between compute cost and object detection accuracy while under various contention scenarios.[1] The algorithm maintains an internal list of possible configurations with corresponding accuracies. When deployed, the framework will then swap out the configuration based on the current contention. One parameter in the possible configurations include how many tracker frames should follow an object detection frame; it is here that a problem occurs when under very high contention scenarios: the object detection algorithm blocks the tracker algorithm which in turn results in dropped frames. Although the Virtuoso framework was made to deal with various contention scenarios, we find the serial nature of the algorithm holds it back when dealing with high contention scenarios.

Our contribution in this paper was to modify the Virtuoso framework to allow relatively high compute cost object detection algorithms without blocking the tracker. This was possible by adding parallelization to the Virtuoso framework by putting the detection algorithm and the tracker algorithm in separate threads. The separation prevents the high compute cost detector algorithm from blocking the tracker algorithm, and so the parallel version of Virtuoso is able to report bounding boxes on every frame even in high contention scenarios.

2. RELATED WORK

Video object detection aims to locate and classify object instances frames of images given at a frame rate. In contrast to single image object detection, video object detection introduces a temporal variable where subsequent

Further author information: (Send correspondence to V.Dasari)
E-mail: venkateswara.r.dasari.civ@army.mil

frames often contain similar information. Therefore a common method is to combine a detection backbone (such as Faster R-CNN[2] or YOLO[3]) with an intelligent way to handle subsequent frames. Recent trends in video object detection focus on creating more complex networks for enhanced accuracy. Techniques like frame aggregation enhance detection by utilizing features from multiple frames. Innovations like SELSA[4] and MEGA[5] have expanded the scope of frame selection, using semantic similarities across different videos. REPP,[6] on the other hand, post-processes detection outputs for improved results. Complementary to these are approaches leveraging optical flow[7] and knowledge distillation.[8] However, these advancements often overlook the efficiency in network size, energy consumption, and latency, critical for real-time applications on computationally constrained platforms like mobile devices. To address this, studies have aimed to balance accuracy and latency. For instance, Feichtenhofer et al.[9] combined object detection with tracking for a more efficient design. Others, like Jiang et al.,[10] used LSTM modules to reduce computation costs, and Chen et al.[11] employed the "key frame" concept to minimize redundant computations.

Efforts to design efficient video object detection systems for mobile devices can try to simply reduce the detection computation necessary [12,13], or dynamically adjust input parameters such as image size such as AdaScale.[14] GhostNet[15] introduces a "ghost module" for feature reuse, lessening computational demands. Additionally, neural architecture search (NAS) techniques, as seen in EfficientDet,[16] have been used to automate the optimization of model architectures, seeking a balance between lightweight designs and performance.

Algorithms designed for embedded real-time systems such as autonomous driving must adhere to latency requirements to maximize safety and functionality. In this context methods such as[17] show that prior benchmarking in GPU hardware can allow for a three way optimization framework that takes into account energy usage, latency requirements, and accuracy requirements. However, this work can fall short, especially in high-contention scenarios, because of the serial nature of the frame processing pipeline. The frames that are used for tracking the ouput of the detection frames waits for the detection backbone to finish, which causes a lag in the overall latency of the system. This work aims to assess the viability of processing detection frames and tracker frames in parallel to reduce the overall latency of the system, while maintaining requisite accuracy.

3. METHODS

3.1 Serial Virtuoso

The Virtuoso framework is an adaptive object detection algorithm. The framework uses a detection algorithm followed by a number of tracking frames. The adaptive part is implemented by using an adaptive configuration that can change the detection algorithm used, the down-sampling factor used to down-sample the input image to the tracker, and an Si parameter which is just the number of tracking frames that should follow each detection frame. The optimal parameters for each contention are found before deployment by using benchmarks to measure compute times and accuracy. Then in the deployment phase the required average sample rate or the required accuracy can be used to filter the possible configurations, and from the available configurations the optimal parameters are chosen. By allowing many possible configurations, the Virtuoso framework can choose the optimal parameters for every contention scenario.

3.2 Parallel Virtuoso

The parallel version of Virtuoso is still an adaptive object detection algorithm, but as can be seen in figure 1 the algorithm is parallelized by separating the detection algorithm and the tracker algorithm into their own threads. The two threads allow the use of a high-cost detection algorithm that can be used to periodically update the tracker thread. The use of a high cost detection algorithm inherently causes a large time delay between the detection frame and the first use of the tracker, so the detection thread also uses the results of the detection algorithm to initialize a tracker which is then warmed up by applying the tracker to cached frames until the tracker is up-to-date to the latest frame. Once the new tracker is warmed up, the tracker is placed into a cache that the tracker thread will use to swap out to the latest tracker. By separating the tracker and detector into their own threads, we prevent the detector from blocking the tracker which means the parallel version of Virtuoso will report bounding boxes on nearly every frame even when using relatively high cost detection algorithms in high contention scenarios.

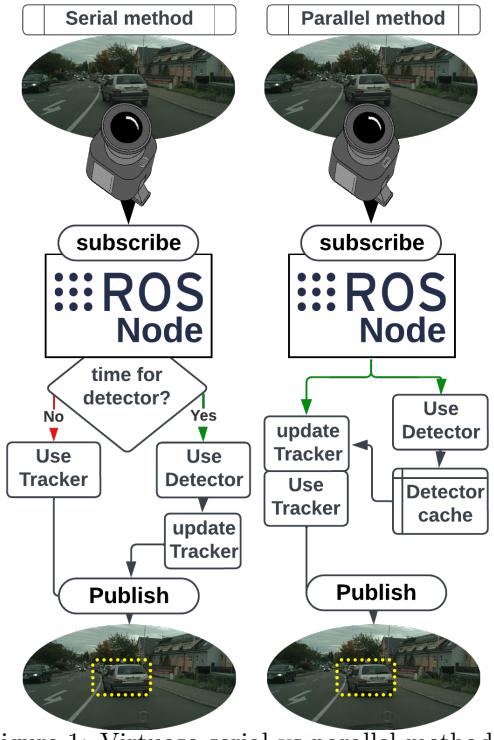

Figure 1: Virtuoso serial vs parallel methods.

Algorithm 1: Parallel Detector

```
image←published_image;
bbox←run_detector(image);
detector_cache.append(bbox);
```

Algorithm 2: Parallel Tracker

```
while image_published do
    if len(detector_cache) >0 then
        bbox←run_detector();
        reset_tracker()
    else
        bbox←tracker.update_tracker()
    end
    set_prev_frame(bbox)
end
```

4. TECHNICAL APPROACH

The data is collected in two parts with the first being a test of the available trackers and the second being a test of the serial method compared to the parallel method. The data for the tracker test is to determine which of the trackers packaged with OpenCV provide the best results when applied to our field data. Once the trackers are chosen we can then collect data to compare the serial method to the parallel method; specifically we want to know if it is better to choose a light weight detector frame followed by tracker frames for the serial method, or if it's better to use a parallel method where we choose a robust detector algorithm that periodically updates a separate tracker thread.

4.1 Tracker Test

The tracker test is to determine the optimal trade-off between compute time and accuracy amongst the available trackers. Initially we just qualitatively compared the trackers by changing the tracker type and started up a ROS-bag to see the change in accuracy. Even with the qualitative comparison, we found many of the trackers were insufficient for our tracker task. What usually happened is the trackers would perform well when there was no contention, but then the trackers would start to fail when contention on the GPU was increased. It should be noted the trackers run on the CPU, so contention on the GPU meant the inference time of the detection algorithm was long enough that multiple frames would be tossed out between the initialization of the tracker to when the tracker was first used. We believe the dropped frames between initialization to the first time the tracker was used caused the drop in tracker accuracy. We also found the compute time of each tracker would cause some frames to be skipped even during the tracker phase of both the serial and parallel methods; these skipped frames would generally reduce the tracker accuracy. To quantify our frustration we perform both an accuracy and computation time comparison of the trackers.

For the accuracy test we wanted to quantify the drop in accuracy as the number of skipped frames between each tracker frame is increased. To generate the data, we took our video stream labeled with truth boxes and initialized a new tracker on every frame using the truth box for initialization to the tracker. Then we allowed each tracker to run on subsequent frames taking into account the tracker should not be applied on the skipped frames. The accuracy data for all trackers was accumulated relative to the start frame of each tracker; for example,

tracker-50 might have been initialized on frame 50 and tracker-60 might have been initialized on frame 60, but using a skip value of 2 would mean the accuracy for tracker-50 on frame 52 and the accuracy for tracker-60 on frame 62 would be accumulated to the same data point. Once the data is accumulated for each skip value, we can measure the drop in accuracy due to missing the first N frames after using the detection frame to initialize the tracker, and we can also measure the drop in accuracy due to the tracker itself using too much compute time to be applied to every frame. Once we know the relative tracker accuracies, we can choose the two best trackers that should work even in high contention scenarios.

For the compute time measurement we compare the compute costs for the two most viable trackers. The compute time is measured by first choosing a random frame from the video stream and scaling the image to a particular up-sampling factor, then choosing a random box in the image with a given box size, and finally tracking that box on the very next frame. Given the two independent variables of up-sampling factor and box size, we can measure the compute time for the two most viable trackers and determine which tracker we want to use for our serial vs parallel method test.

4.2 Serial vs Parallel Virtuoso

The serial vs parallel method comparison requires testing each method under various contention scenarios and then for each scenario checking which method gives the best metrics. The contention scenario is created by just adding a time.sleep after every pass of the detection algorithm that is proportional to the inference time of the detection algorithm; this mimics contention on the GPU as the detection algorithm runs on the GPU while the tracking algorithm runs on the CPU. For each contention a brute-force configuration search is done for each method where we vary the parameters of each method to find the parameters that give the optimal metric.

For the serial method we can perform a brute force parameter search by changing the detection algorithm used, the down-sampling factor used for the input to the tracker, and the number of tracker frames used after every detection frame. Here we have a total of 3 parameters to vary, which means there are a total of 125 possible configurations per contention scenario when we limit each parameter to only 5 possibilities. We used a total of 6 contention scenarios for a total of 725 simulations.

For the parallel method we can perform a brute force parameter search by changing the detection algorithm used and the down-sampling factor used for the input to the tracker. This search uses one parameter less than the serial method, so with 6 contention scenarios we have only 125 simulations for the parallel method. It should be noted that we modified the down-sampling selection for the parallel method. The original down-sampling method used a single down-sampling factor for all trackers, but the down-sampling method in the parallel method was modified to allow per box down-sampling that selected the least down-sampling factor among [0.5,1,2,4,8] where the bounding box had less pixels than some determined PIX value. This per box down-sampling selection helped with algorithms that have a compute time that depends on the tracked box size.

For each simulation the ROS-core is running while the field data is being published as a ROS-topic; the method is also deployed as a ROS-topic that subscribes to the field data topic and publishes the predicted bounding boxes as a string topic. The predicted data is recorded using rosbag record and is analyzed once all simulations are done. For simplicity, we set the detection algorithm confidence threshold to 30% for all simulations.

To determine which simulation is the best for a given contention scenario we need a metric that punishes both false positives and false negatives. A good metric would have been to generate the precision vs recall curve by varying the detection algorithm confidence threshold, but our simulations have a constant 30% confidence threshold. Instead we estimate the AUC using a single data point as outlined in equation 1 which we call AUC* as this metric at least shares the same units as AUC. By finding the simulation with the best AUC* we can find the best parameters for each contention scenario.

$$
\begin{aligned}
TP &= TP_{car} + TP_{human} = \text{total true positives of car+human} \\
FP &= FP_{car} + FP_{human} = \text{total false positives of car+human} \\
FN &= FN_{car} + FN_{human} = \text{total false negatives of car+human} \\
precision &= TP/(TP + FP) \\
recall &= TP/(TP + FN) \\
AUC* &= precision * recall
\end{aligned} \tag{1}
$$

5. RESULTS

5.1 Tracker Test Results

The tracker accuracy test results can be seen in figure 2 where the accuracy vs the time from initialization is plotted for various tracker frame rates. When the tracker frame rate is equal to the camera frame rate we find most trackers do well for anything less than 10 frames; this means most of the trackers do well when tracking for less than a third of a second. However; the CSRT tracker is much better than most trackers past 30 frames; this means a tracking task that last for a second or more should use the CSRT tracker. The interesting part is that as the frame rate of the tracker is reduced the accuracy of the CSRT tracker and the Medianflow tracker are nearly identical. For this reason we can consider CSRT and Medianflow to be the two best trackers when experiencing high contention scenarios.

The two plots in figure 3 show the compute time measurements for CSRT and Medianflow as a function of the two independent variables up-sample factor and box-size. It can be seen that CSRT has over twice the compute cost, and it can also be seen that CSRT has a box-size dependence while Medianflow does not. At the moment we are considering high GPU contention scenarios where the initial skip factor is high, so we chose the Medianflow tracker for all our simulations.

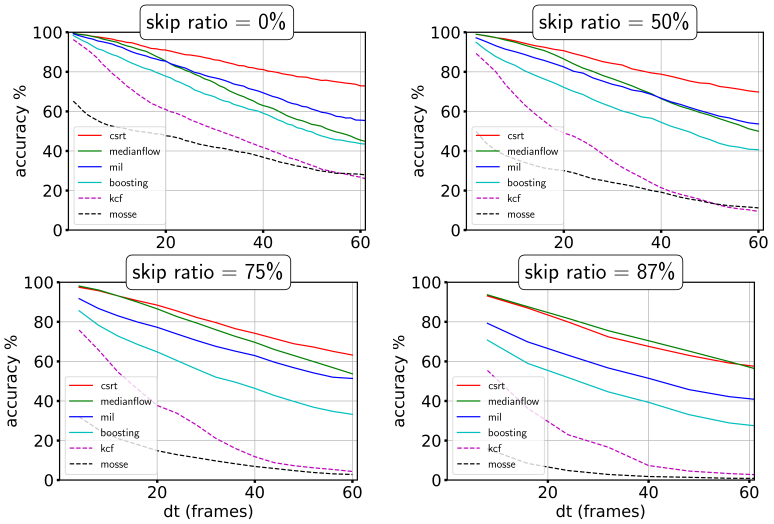

Figure 2: Tracker accuracy plotted as a function of the time since the original frame was received. The skip ratio represents what percentage of frames are skipped between tracker frames.

5.2 Serial vs Parallel Virtuoso Results

The quantitative results for the serial vs parallel test are in figure 4 and table 1. The figure plots AUC* for both methods; this figure shows the parallel method has a higher AUC* than the serial method for all contention scenarios. The table lists the corresponding optimal parameters for every contention scenario for both methods; this table shows the serial method is initially better off decreasing the compute cost of the detection algorithm

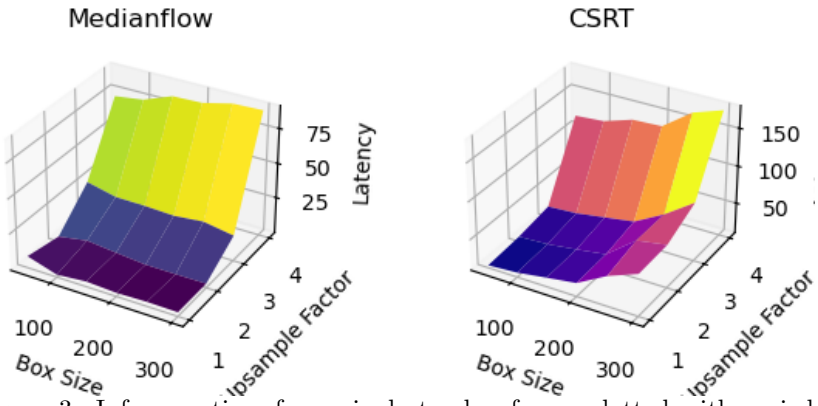

Figure 3: Inference time for a single tracker frame plotted with varied box size and upsampling factor for the input image.

Table 1: AUC* (%) to compare the parallel method to the serial method. The parallel method will use downsample factors(DS) in [0.5,1,2,4,8] until the initial tracked box contains less than PIX pixels. The Si parameter is the number of tracker frames after every detection frame for the Serial method. The YOLOv8 version of YOLO is used.

	Contention Factor					
	0	**1**	**3**	**7**	**12**	**15**
Parallel parameters	auc=87.1 PIX=1600 YOLO-XL	auc=86.8 PIX=1600 YOLO-XL	auc=83.0 PIX=1600 YOLO-L	auc=78.5 PIX=400 YOLO-M	auc=76.9 PIX=100 YOLO-M	auc=71.9 PIX=400 YOLO-M
Serial parameters	auc=86.5 DS=2 Si=1 YOLO-L	auc=64.8 DS=2 Si=1 YOLO-S	auc=38.6 DS=8 Si=1 YOLO-Nano	auc=25.0 DS=1 Si=16 YOLO-Nano	auc=19.3 DS=1 Si=12 YOLO-Nano	auc=12.6 DS=1 Si=16 YOLO-Nano

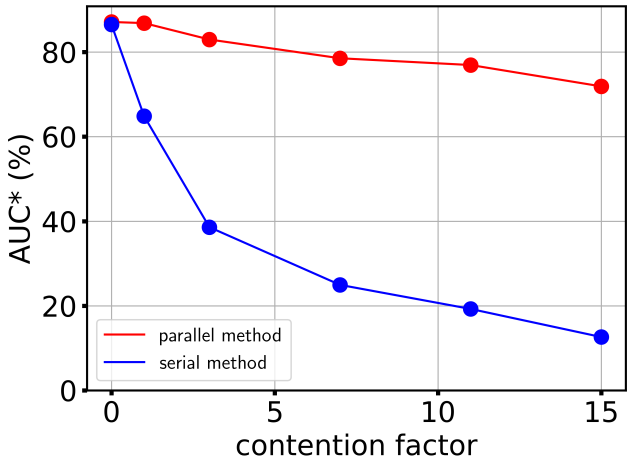

Figure 4: The AUC* plotted for the serial/parallel method as the contention factor is increased. Note AUC* was estimated using a single confidence threshold of 30% for the yolo detector.

as the contention factor is increased, but eventually the serial method is using the smallest detector algorithm which forces the serial method to then increase the number of tracker frames used after the detection frame; i.e. the Si parameter for the serial method increases after the smallest detector algorithm is chosen. By contrast the parallel method generally uses a higher compute cost detector algorithm, but initially the parallel algorithm

Figure 5: qualitative comparison of the serial and parallel method for contention factor=15.

does decrease from YOLO-XL down to YOLO-M before the parallel method decides to start down-sampling the image via the PIX parameter. Although both the serial and parallel method give similar results for low contention scenarios, the parallel method performs significantly better for high contention scenarios.

We believe the parallel method performs better in high contention scenarios because the parallel method uses a higher compute cost detection algorithm as well as initializing and warming up the tracker while still in the detector thread. The higher compute cost tracker allows for more consistent object detection which ensures the tracker thread always has a relatively confident true positive to track. Also, putting the tracker warm up phase inside the detector thread prevents the tracker thread from being blocked while the next tracker is generated. One example of the serial method failing while the parallel method succeeds can be seen in figure 5 which shows qualitative output for the scenario with contention factor 15; here the serial method initially detected a human, but the human was lost in a subsequent detector frame where either the detector did not detect the human or the detection itself was blocking so long that frames were dropped. By contrast the parallel method in figure 5 successfully detected the human for all frames while using a detector algorithm that cost more than the serial method. By using separate threads for the detection and tracker tasks, the parallel method was able to consistently return bounding boxes even in high contention scenarios.

6. CONCLUSION AND FUTURE WORK

Our research set out to demonstrate the parallel version of Virtuoso was a viable alternative to the serial version of Virtuoso. Specifically in high contention scenarios we hypothesized the parallel method should be able to achieve better performance than the serial method as the parallel method can process tracking requests on every frame. However, we encountered challenges in obtaining positive results with the parallel method. One challenges was the parallel method depended on the accuracy of the trackers. Counter-intuitively, using what some would consider "better" trackers resulted in failure, which prompted us to conduct a tracker test. The tracker test revealed that only two trackers were suitable for the parallel method. This finding was significant because it showed the choice of tracker was critical for the success of the parallel method. Another challenge was there existed a random chance for a tracker to fail when the time elapsed between initialization and first use was excessive; this was a problem because we aimed to use higher compute cost detectors in the parallel method which inevitably caused a substantial delay. To mitigate this delay, we implemented a warm-up phase for the tracker. The warm-up phase allowed the tracker to be initialized using both the detection frame and used on subsequent frames that would have normally been cleared from the cache; this resulted in significantly less tracking failures. Despite the challenges, our results revealed that the parallel method exhibited a preference for higher compute cost detector algorithms compared to the serial method. This preference enabled the parallel method to outperform the serial method in high contention scenarios. Our research thus provides empirical evidence supporting the feasibility of the parallel method as an alternative approach in high contention environments. In conclusion, our research has demonstrated that the parallel version of Virtuoso can be a viable alternative to the serial version of Virtuoso in high contention scenarios provided the proper trackers are used and also warmed up before use.

Our future work includes a more in-depth analysis to the failure modes of the current parallel method as well

as looking into the possibility of using multiple trackers to increase the tracker accuracy. Currently the trackers can detect failure to track events, but at the moment failed trackers are just dropped; instead we want to try a recovery algorithm that re-initializes a different tracker that is subsequently warmed up to the current frame. We also want to try using the more accurate CSRT tracker for the warm-up phase and then switching to the lower compute cost Medianflow tracker for the tracking thread.

ACKNOWLEDGMENTS

The authors thank AI for Maneuver and Mobility(AIMM) Essential Research Program at DEVCOM US ArmyResearch Laboratory and High-Performance Computing Internship Program (HIP) for supporting this research project.

REFERENCES

[1] Wang, Y., Wang, W., Liu, D., Jin, X., Jiang, J., and Chen, K., "Enabling edge-cloud video analytics for robotics applications," *IEEE Transactions on Cloud Computing* (2022).
[2] Ren, S., He, K., Girshick, R., and Sun, J., "Faster r-cnn: Towards real-time object detection with region proposal networks," *Advances in neural information processing systems* **28** (2015).
[3] Redmon, J. and Farhadi, A., "Yolov3: An incremental improvement," *arXiv preprint arXiv:1804.02767* (2018).
[4] Wu, H., Chen, Y., Wang, N., and Zhang, Z., "Sequence level semantics aggregation for video object detection," in [*Proceedings of the IEEE/CVF International Conference on Computer Vision*], 9217–9225 (2019).
[5] Chen, Y., Cao, Y., Hu, H., and Wang, L., "Memory enhanced global-local aggregation for video object detection," in [*Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*], 10337–10346 (2020).
[6] Sabater, A., Montesano, L., and Murillo, A. C., "Robust and efficient post-processing for video object detection," in [*2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*], 10536–10542, IEEE (2020).
[7] Zhu, X., Wang, Y., Dai, J., Yuan, L., and Wei, Y., "Flow-guided feature aggregation for video object detection," in [*Proceedings of the IEEE international conference on computer vision*], 408–417 (2017).
[8] Deng, J., Pan, Y., Yao, T., Zhou, W., Li, H., and Mei, T., "Relation distillation networks for video object detection," in [*Proceedings of the IEEE/CVF international conference on computer vision*], 7023–7032 (2019).
[9] Feichtenhofer, C., Pinz, A., and Zisserman, A., "Detect to track and track to detect," in [*Proceedings of the IEEE international conference on computer vision*], 3038–3046 (2017).
[10] Jiang, Z., Liu, Y., Yang, C., Liu, J., Gao, P., Zhang, Q., Xiang, S., and Pan, C., "Learning where to focus for efficient video object detection," in [*Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XVI 16*], 18–34, Springer (2020).
[11] Chen, K., Wang, J., Yang, S., Zhang, X., Xiong, Y., Loy, C. C., and Lin, D., "Optimizing video object detection via a scale-time lattice," in [*Proceedings of the IEEE conference on computer vision and pattern recognition*], 7814–7823 (2018).
[12] Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C., "Mobilenetv2: Inverted residuals and linear bottlenecks," in [*Proceedings of the IEEE conference on computer vision and pattern recognition*], 4510–4520 (2018).
[13] Zhang, X., Zhou, X., Lin, M., and Sun, J., "Shufflenet: An extremely efficient convolutional neural network for mobile devices," in [*Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*], (June 2018).
[14] Chin, T.-W., Ding, R., and Marculescu, D., "Adascale: Towards real-time video object detection using adaptive scaling," *Proceedings of machine learning and systems* **1**, 431–441 (2019).
[15] Han, K., Wang, Y., Tian, Q., Guo, J., Xu, C., and Xu, C., "Ghostnet: More features from cheap operations," in [*Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*], 1580–1589 (2020).
[16] Tan, M., Pang, R., and Le, Q. V., "Efficientdet: Scalable and efficient object detection," in [*Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*], 10781–10790 (2020).
[17] Lee, J., Wang, P., Xu, R., Jain, S., Dasari, V., Weston, N., Li, Y., Bagchi, S., and Chaterji, S., "Virtuoso: Energy-and latency-aware streamlining of streaming videos on systems-on-chips," *ACM Transactions on Design Automation of Electronic Systems* **28**(3), 1–32 (2023).