Band: Coordinated Multi-DNN Inference on Heterogeneous Mobile Processors

Joo Seong Jeong*
Seoul National University
joosjeong@snu.ac.kr

Jingyu Lee*
Seoul National University
jingyu.lee@hcs.snu.ac.kr

Donghyun Kim
Seoul National University
dh.kim@snu.ac.kr

Changmin Jeon
Seoul National University
wisechang1@snu.ac.kr

Changjin Jeong
Seoul National University
create.jeong@snu.ac.kr

Youngki Lee†
Seoul National University
youngkilee@snu.ac.kr

Byung-Gon Chun†
Seoul National University,
FriendliAI
bgchun@{snu.ac.kr,friendli.ai}

ABSTRACT

The rapid development of deep learning algorithms, as well as innovative hardware advancements, encourages multi-DNN workloads such as augmented reality applications. However, existing mobile inference frameworks like TensorFlow Lite and MNN fail to efficiently utilize heterogeneous processors available on mobile platforms, because they focus on running a single DNN on a specific processor. As mobile processors are too resource-limited to deliver reasonable performance for such workloads by their own, it is challenging to serve multi-DNN workloads with existing frameworks.

This paper introduces BAND, a new mobile inference system that coordinates multi-DNN workloads on heterogeneous processors. BAND examines a DNN beforehand and partitions it into a set of subgraphs, while taking operator dependency into account. At runtime, BAND dynamically selects a schedule of subgraphs from multiple possible schedules, following the scheduling goal of a pluggable scheduling policy. Fallback operators, which are not supported by certain mobile processors, are also considered when generating subgraphs. Evaluation results on mobile platforms show that our system outperforms TensorFlow Lite, a state-of-the-art mobile inference framework, by up to 5.04× for single-app workloads involving multiple DNNs. For a multi-app scenario consisting of latency-critical DNN requests, BAND reaches up to 3.76× higher SLO satisfaction rate.

CCS CONCEPTS

• **Human-centered computing** → **Ubiquitous and mobile computing systems and tools**; • **Computer systems organization** → **Real-time system architecture**.

*Both authors contributed equally to the paper
†Corresponding authors

MobiSys '22, June 25–July 1, 2022, Portland, OR, USA

ACM ISBN 978-1-4503-9185-6/22/06…$15.00
https://doi.org/10.1145/3498361.3538948

KEYWORDS

Mobile Deep Learning, Heterogeneous Processors, DNN Accelerators, Multi-DNN Inference

ACM Reference Format:
Joo Seong Jeong, Jingyu Lee, Donghyun Kim, Changmin Jeon, Changjin Jeong, Youngki Lee, and Byung-Gon Chun. 2022. Band: Coordinated Multi-DNN Inference on Heterogeneous Mobile Processors. In *The 20th Annual International Conference on Mobile Systems, Applications and Services (MobiSys '22), June 25–July 1, 2022, Portland, OR, USA*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3498361.3538948

1 INTRODUCTION

Thanks to the rapid development on deep learning algorithms, recent mobile applications leverage a wide variety of deep neural networks (DNNs) to solve various tasks such as person identification [57] on camera frames and live video analytics [60]. Augmented Reality (AR) applications process many DNNs in a single frame [57, 60] or require concurrent support of tasks, each with its own service-level objectives (SLOs) for real-time tasks like hand pose estimation and object tracking [9, 31].

Unfortunately, current state-of-the-art mobile deep learning frameworks, including TensorFlow Lite [8], MNN [29], Mace [7], and NCNN [4], are not quite fit to serve mobile applications, because they are developed and optimized toward executing a single DNN as fast as possible. diffAlthough modern mobile devices come equipped with multiple processors for DNN computations such as the Edge TPU [5], the Hexagon DSP [11], and the Adreno GPU on Google Pixel 4, frameworks designate a fast processor and use only that processor to run a DNN. They lose out on opportunities to utilize all processors when serving multiple inference requests. It is technically possible to run multiple DNNs on one processor, but mobile processors are far too resource-limited to deliver reasonable performance (10+ frames per second) for multi-DNN workloads on their own.

In this paper, we present a mobile inference system, BAND, that coordinates multi-DNN workloads on heterogeneous processors. BAND is able to dynamically devise DNN execution plans and schedule DNNs on processors, according to given scheduling goals such as least slack time or makespan minimization. BAND differs from existing frameworks in that it can prioritize certain DNNs over others to reach the scheduling goal, and even schedule DNNs in a smaller granularity of subgraphs – processing up to 16.8 frames per second for complicated multi-DNN workflows.

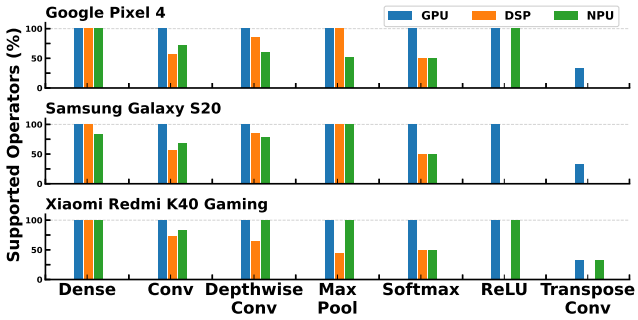


Figure 1: Percentage of supported operators across various DNNs and devices. Even within the same category, operators may have varying support (e.g., 1x1 convs are supported, but strided or dilated convs are not).

BAND faces several challenges when coordinating multiple DNNs on heterogeneous processors. First, BAND must take care to avoid processor contention from scheduling DNNs on the same processor. Especially, the system must take the varying performance characteristics of heterogeneous processors into account.

Second, due to the heterogeneity of processors in mobile environments, BAND must deal with *fallback operators* that are unsupported on certain processors. As shown in Figure 1, many DNN operators are not fully supported on recent mobile processors. While executing a DNN with such operators on a processor (e.g., NPU), the system must fall back to the CPU in order to run those operators. During this fallback, BAND can run another model's operators on that processor to take full advantage of the processor's resources. This opens up the possibility of creative schedules with high processor utilization.

Third, BAND must consider the performance fluctuations of mobile processors when determining DNN schedules. The dynamic voltage and frequency scaling (DVFS) mechanisms on mobile SoCs obstruct accurate latency prediction, which is crucial in scheduling DNNs on processors.

A handful of research efforts [21, 24–26, 42, 48] have attempted to run multiple DNNs on one or more processors. However, they assume certain task requirements and provide task-specific solutions. Moreover, none of them consider the coordination of fallback operators, which is commonly present in mobile environments. Some works also propose offloading to edge infrastructures, but we do not consider offloading as the network latency is usually too large to satisfy the service-level requirements of modern mobile workloads.

To address the challenges, we build BAND with a central model analyzer and scheduler component for coordinating multiple DNNs on heterogeneous processors. BAND partitions and schedules DNNs in granularity of *subgraphs*. In fact, BAND prepares a diverse set of subgraphs per DNN, such that a DNN can be executed in multiple ways. At runtime, BAND's scheduler dynamically decides which subgraphs to use to run the DNN. This grants BAND flexibility in scheduling multiple DNNs, reinforcing the system's coordination capabilities. Fallback operators are considered when generating subgraphs to avoid processor contention from silent fallbacks at runtime. Furthermore, BAND adopts global queueing, scheduling one subgraph (per processor) at a time to mitigate the limitations of scheduling tasks on non-preemptible mobile processors with unpredictable runtime fluctuations.

We evaluated BAND on recent mobile devices and compared its performance with TensorFlow Lite. For single-app workloads that involve multiple DNNs, BAND outperforms TensorFlow Lite by up to 5.04×. BAND also achieves superior SLO satisfaction rates of up to 3.76× for a multi-app workload consisting of latency-critical DNNs, where each app independently issues requests periodically.

Our contributions are summarized as follows:

- We propose BAND, a mobile DNN runtime for multi-DNN requests utilizing heterogeneous processors on a mobile device. To our knowledge, BAND is the first mobile platform which can support general multi-DNN workloads from multiple mobile applications.
- We design a novel DNN partitioning and scheduling mechanism based on the operator support for heterogeneous processors. Together with this mechanism and pluggable scheduling policies, our system can satisfy different requirements from various applications.
- We implement BAND on TensorFlow Lite [8], and evaluate our system with different application requirements. Various DNN models and mobile devices are evaluated to show the effectiveness of our system.

2 WORKLOAD CHARACTERISTICS

This paper targets multi-DNN tasks from mobile applications. Such tasks utilize DNNs to interpret various sensory inputs, capturing user contexts as well as the surroundings. Multi-DNN tasks can be imposed to (i) analyze a single sensing input in various forms (e.g., face detection, recognition, and pose estimation over video streams) [57], (ii) analyze multiple sensing modalities (e.g., audio-visual AR) [36, 59], and (iii) handle concurrent sensing applications [35, 41]. In this section, we describe the unique workload characteristics of such multi-DNN tasks.

Service requirements. Multi-DNN tasks from mobile apps have two dominant service requirements, with varying expectations of responsiveness. The most common type is a minimization of *makespan*, which specifies the completion time of a set of tasks that must all be finished to correctly analyze a single unit of sensory input. As an example, mobile apps utilize complex pipelines of task-specific DNNs for detailed analysis of a camera frame. The EagleEye [57] scenario introduces a sequence of DNN tasks (face detection, super-resolution, and face recognition) to identify the faces of people on a video frame. In such a case, the app expects the system to minimize the makespan as users receive feedback for every processed frame.

Another aspect is that mobile applications require timely responses to their *latency-critical* tasks. A DNN task involved in a rendering pipeline of a foreground app requires consistent latency guarantees. For instance, a recent AR application requires at most 20 ms of motion-to-photon latency to avoid simulation sickness [31]. Any delay in a DNN-based hand tracking task for AR glasses such as the Oculus Quest [9] would cause a discontinuity in user experience. Also, any trigger-based sensing app that a user initiates, such as drowsiness detection in navigation apps, requires consistent latencies of under ∼1 second (considering the reaction time under the two-second rule [45]) to give enough time for users to react [47].

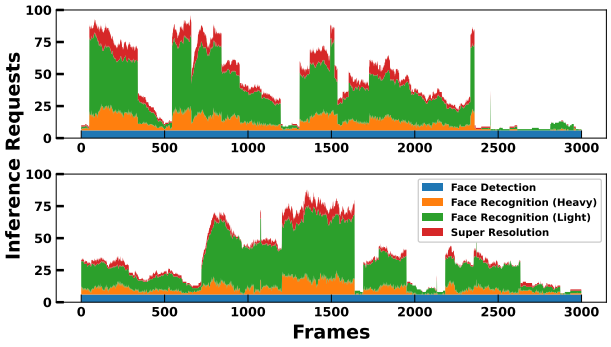


Figure 2: Workload requirements related to scene complexity. The number of required inference requests for the EagleEye Person Finder [57] scenario is shown, based on two different traces from the YouTube Faces dataset [53].

Figure 3: Assembly assistant multi-app scenario. The smartphone simultaneously runs an assembly AR app [13] with heart-rate variability sensing [27] for factory workers.

Dynamic workloads. As DNNs interpret user contexts and surroundings, the overall workload complexity reflects the changes around the user's environment. For example, recent AR apps either have a context-aware pipeline that adaptively selects execution paths, or requires varying computations depending on the scene complexity. Figure 2 shows the dynamic computational requirements of the EagleEye scenario [57]; as the number of faces in a video drastically changes per frame, the number of required inferences also changes. The Distream [60] workflow also involves high workload fluctuations depending on the count and type of objects in live video analytics apps.

Multiple applications. DNNs bring new opportunities for various sensing modalities in smartphones [35] such as audio-based emotion recognition for life-logging [36, 46] and IMU-based driving condition tracking [59]. Similar to conventional apps, future mobile platforms can continuously execute different combinations of foreground and background sensing apps.

Figure 3 shows an assembly assistant AR scenario for factory workers. The smartphone processes video frames to detect the type and orientation of assembly components, based on object detection and pose estimation DNNs. It overlays the proper assembly instructions on the screen. An additional background sensing app [27] can assess the worker's heart-rate variability (HRV) based on front-camera frames, which can detect any sign of long-term exposure to air pollution without regular health checks [44].

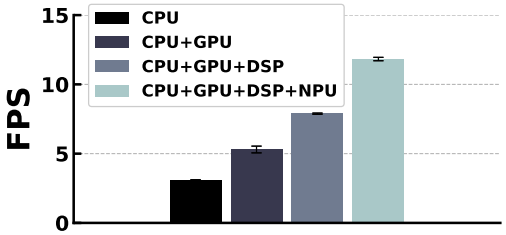


Figure 4: BAND's performance of using heterogeneous processors. The processed frames per second (FPS) rise as more processors are used. Results are from the EagleEye [57] workload, on Google Pixel 4.

3 MOTIVATION

3.1 Utilizing Heterogeneous Processors

Today's mobile deep learning frameworks [4, 7, 8, 29] cannot fully address the requirements of multi-DNN workloads. As frameworks mostly focus on running a single DNN as fast as possible, they only utilize a specific processor such as the GPU or NPU. However, unlike server/cloud environments where GPUs are dominant in terms of floating-point operations per second (FLOPS), there is no de-facto processor for mobile platforms. Therefore, it is preferable to utilize the various processors available on mobile platforms, rather than relying on a single outstanding processor.

Another line of works propose offloading to edge infrastructures as an alternative solution to serve modern workloads [19, 30, 37]. As data centers provide substantially stronger computing power than mobile processors, it may be favorable to offload computation to servers. However, this paper targets workloads with tight service requirements (10+ FPS), and recent studies show that the latency of offloading heavy computation to edge or cloud infrastructures is generally too large (more than 100 ms) to meet such requirements [55]. Moreover, data privacy is a constantly raised issue regarding offloading. Thus, we do not consider offloading in this paper, and instead, only work with the resources available on mobile devices.

Figure 4 depicts a preview of the performance of our system, BAND. BAND is able to utilize heterogeneous processors to serve EagleEye [57], a multi-DNN workload. Using only the CPU and GPU yields a frame rate of 5.3 frames per second (FPS), but adding the DSP and NPU boosts this number to 11.8 FPS – a 2.23× increase. This point highlights the need of handling multiple processors to serve workloads consisting of multiple DNNs.

3.2 Challenges in Coordination

3.2.1 Processor Contention. It is well known that processor contention occurs when multiple DNNs try to utilize the same processor. This is especially a problem for mobile processors, due to the limited number of processor cores and memory bandwidth. Table 1 shows the latency of a single InceptionV4 model, when multiple model instances are running on the same processor. For most processors, the latency rapidly rises as the number of models is increased. Moreover, SDKs for mobile processors like NNAPI [3] do not provide interfaces for concurrent execution, unlike software libraries provided for acceleration on server GPUs such as NVIDIA CUDA [2] (CUDA streams). The MediaTek APU 3.0 processor in Table 1 manages to

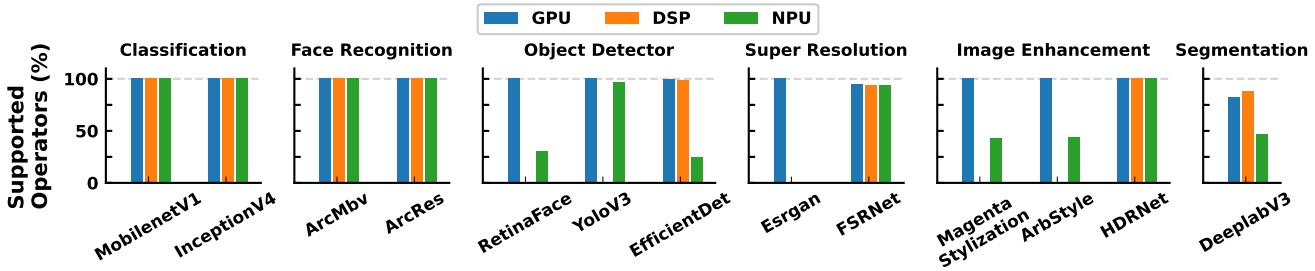


Figure 5: Fallbacks commonly occur across various tasks. Percentage of supported operators on Google Pixel 4 is shown.

Processor (Mobile Device)	The Number of Concurrent Models 1	2	4
Google Edge TPU (Google Pixel 4)	**25.29** ± 0.79	**35.86** ± 10.23	**56.94** ± 22.55
Hexagon DSP (Google Pixel 4)	**25.43** ± 0.55	**37.34** ± 11.41	**61.32** ± 25.08
Qualcomm HTA (Samsung Galaxy S20)	**23.98** ± 1.95	**24.08** ± 2.68	**33.66** ± 10.47
Adreno 650 GPU (Samsung Galaxy S20)	**115.52** ± 0.97	**228.33** ± 3.16	**448.34** ± 7.47
MediaTek APU 3.0 (Xiaomi Redmi K40 Gaming)	**20.34** ± 0.19	**21.35** ± 0.24	**31.61** ± 9.78
Mali-G77 GPU (Xiaomi Redmi K40 Gaming)	**133.36** ± 2.22	**255.49** ± 5.52	**477.91** ± 37.08
Huawei NPU (Huawei Mate 40 Pro)	**10.15** ± 0.14	**14.92** ± 4.26	**23.53** ± 9.57

Table 1: Inference latency variation from concurrent inferences. The mean latency and standard deviation (ms, per model) of running InceptionV4 on various processors are shown, for a varying number of concurrent inferences.

run up to 2 models in parallel with little overhead, but this is not nearly enough to support realistic multi-DNN workloads.

> **A coordinated runtime that utilizes heterogeneous mobile processors must avoid processor contention between multiple DNNs.**

3.2.2 Contention from Fallbacks. For mobile devices, a handful of vendors provide their own processors, and each mobile platform is equipped with a unique combination of such processors. This adds a unique heterogeneity aspect to mobile AI accelerators – not all processors are backed with a wide variety of supported ops. A certain operator may or may not run on a certain processor.

This heterogeneity commonly occurs across various task types. Figure 5 shows the percentage of supported operators on Google Pixel 4, across various DNNs from mobile workloads. Operators from classification models that have been prominent in the community for a long time, are generally supported by recent processors. They are widely used as a backbone for other tasks such as face recognition. On the other hand, a considerably high amount of operators from specialized models such as super resolution models and image enhancement models are not supported. For example, FSRNet [15] includes the transposed convolution operator, which cannot be run on the DSP because of its wide stride value of 4.

Unfortunately, the limited operator support is not simply an engineering issue that can totally be resolved by implementing processor-specific kernels. The core components of recent accelerators are hard-wired circuits designed for a limited set of compute-intensive operations. For instance, the systolic array of the Google Edge TPU [5, 56] and the 3D Cube of the Huawei DaVinci architecture [10] exclusively target compute-intensive operators with specific parameters.

Existing frameworks [7, 8, 29] that are only capable of utilizing a single processor at a time silently fall back to the CPU when running unsupported operators; also called *fallback operators*. Even if these frameworks were extended to employ multiple processors, the silent fallback scheme would severely limit the framework's schedulability in utilizing the various processors.

First, an uncoordinated fallback blocks other operators from accessing a processor that is otherwise idle. For example, in the execution timeline of MobileNetV1 and EfficientDet shown in Figure 6, EfficientDet is occupying the NPU. Around $t = 7ms$, EfficientDet's fallback operators start to run on the CPU, and the NPU becomes idle (the hatched NPU area). However, this information is not exploited in scheduling MobileNetV1. MobileNetV1 simply runs on the unfavored CPU (a). An alternative, the better schedule would be to schedule MobileNetV1 on the NPU immediately after EfficientDet's NPU operators (b), reducing up to 58% of inference latency.

Second, other non-CPU accelerators were not considered as an option to process the fallback operators. Existing frameworks always run fallback operators on the CPU, even though they could be supported by other accelerators. If the fallback operators from the hatched area of EfficientDet in Figure 6 can be run on the GPU faster than the CPU, a more preferable schedule would be running the fallback operators on the GPU to achieve shorter latency.

> **Fallback operators must be carefully scheduled to boost schedulability and processor utilization.**

3.2.3 Uncertainties in Performance. When coordinating multi-DNN workloads on multiple processors, another aspect that complicates the coordination is the unpredictableness in performance. Mobile processors are prone to significant performance fluctuation. Unlike server GPUs that provide relatively deterministic performance [23], mobile processors have innate uncertainties in performance due to the design principles of the system on a chip (SoC) architecture.

SoCs are equipped with DVFS mechanisms for balancing the power consumption and performance, by adjusting voltage-frequency

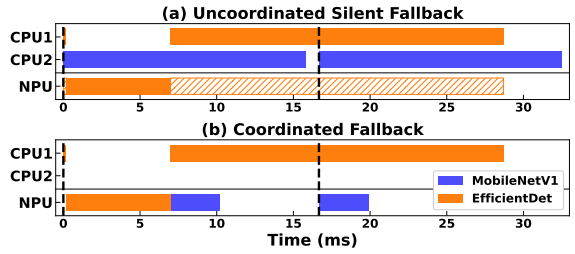


Figure 6: Schedulability limitations of uncoordinated fallbacks. Execution timeline of MobileNetV1 and EfficientDet on the CPU and NPU (Google Pixel 4's Edge TPU). The fallback NPU region of EfficientDet (hatched area) can be used by MobileNetV1 to run faster than on the CPU.

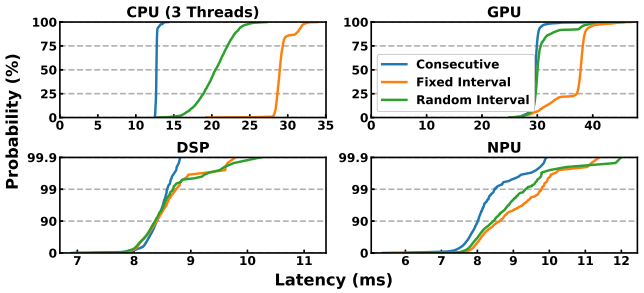


Figure 7: Performance variation of mobile processors. The inference latency of the ArcFace-ResNet50 model on a Google Pixel 4 device is shown as a CDF, with various intervals between inferences.

levels. Although DVFS has its merits, the automatic frequency adjustment makes it difficult for an inference system to estimate the latency of a model.

Figure 7 demonstrates a case where the inference latency of a model fluctuates greatly, up to 2.61×. The latency distribution of the ArcFace-ResNet50 model on a Google Pixel 4 device is shown. In case of the CPU and GPU, when the model is constantly run over and over again ("Consecutive"), DVFS maintains a high processor frequency and the inference latency is small. On the other hand, if we leave the processor idle for long intervals between each inference ("Random" and "Fixed"), the processor frequency keeps fluctuating and the inference latency is relatively large.

Meanwhile, the DSP and NPU are mobile accelerators [11, 28] of separate hardware that are loosely coupled with the main chipset. Thus any computation on the DSP or NPU requires communication through multiple layers of interfaces that causes performance fluctuation, despite being single-frequency processors [14]. It can be observed in Figure 7 that the tail latencies on the DSP and NPU are much higher than the medians, up to 1.53×.

One approach to handle the unpredictableness in mobile processors is preemption. However, mainstream mobile processors do not support context-switching during computation [5, 11]. There exist some academic efforts that experiment with preemption on NPUs, but they incur large performance costs of up to 35% of system throughput for convolutional neural networks [16, 34]. Also, context-switching requires additional stateful logic blocks that could burden the hardware design of mobile SoCs with small form-factors [34, 51], and there is no open interface or control mechanisms exposed to software developers.

> **Uncertainties in mobile platforms must be carefully handled with non-preemptible processors, when coordinating multi-DNN workloads.**

4 BAND: SUBGRAPH-CENTRIC COORDINATION

We now describe our approach to solving the challenges from Section 3. Rather than blindly delegating inferences to processors and expecting them to operate in an efficient manner, we place a central component that devises a fine-grained schedule plan of coordinating multiple DNNs on multiple processors. This way, we can make informed decisions with a global view of the system state.

Moreover, we propose to partition models into *subgraphs*, and run models based on dynamically determined subgraph schedules. Existing mobile deep learning frameworks treat DNN models as unsplittable entities in terms of scheduling. This aspect makes it difficult to react to the dynamics in mobile workloads, such as receiving new inference requests, experiencing latency fluctuations, or encountering fallback operators. Instead, we separate fallback operators from non-fallback operators by partitioning them into different subgraphs. This allows us to coordinate multiple requests in a fine-grained manner and concoct fallback-aware schedules such as the one shown in Figure 6. Moreover, we consider multiple possible schedules of subgraphs at runtime rather than simply following a predetermined static schedule. This way, we proactively react to unpredicted fluctuations by dynamically deciding subgraph schedules.

4.1 System Overview

We describe BAND, a mobile inference system that utilizes heterogeneous processors to serve multi-DNN workloads. BAND offers a coordinated runtime consisting of a model analyzer, a central scheduler, and per-processor workers. The model analyzer partitions models into subgraphs, while the scheduler decides which subgraphs to run on which workers. The workers execute the subgraphs on their respective processors.

Figure 8 illustrates the system architecture of BAND. Client applications first notify BAND of the DNN model that they will use, via ①`RegisterModel`. Upon receiving a model, the model analyzer examines the model and generates subgraphs of the model (Section 4.2). As operators have heterogeneous processor compatibilities, the analyzer carefully groups operators so that the scheduler can have fine-grained control over fallback handling. A subgraph is not mapped to a specific processor at this point; it is up to the scheduler to determine where to run a subgraph, at runtime.

Next, clients issue inference requests via ②`RunModel`. Requests are enqueued into a queue managed by the scheduler in the form of *jobs*. A job is an internal data structure that includes the requested model and input data, as well as request metadata such as SLO value and request app id. Based on the subgraph information from the analyzer, the scheduler (Section 4.3) dequeues a job from its

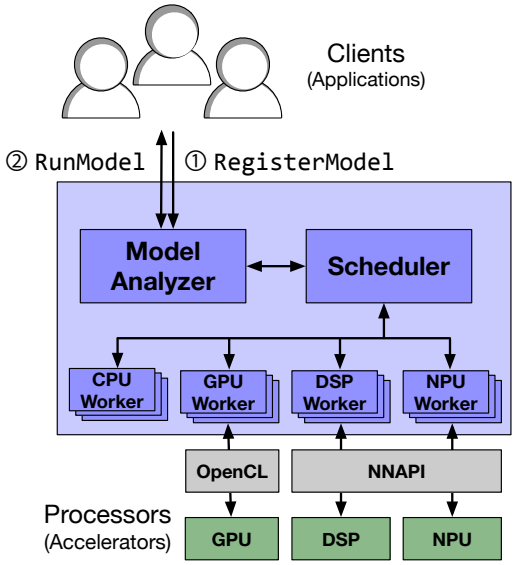
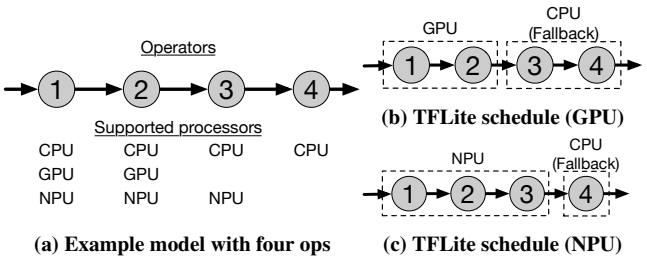


Figure 8: BAND system architecture.



(a) Example model with four ops **(b) TFLite schedule (GPU)** **(c) TFLite schedule (NPU)**

Figure 9: An example model with varying operator support. For a given processor, TensorFlow Lite only creates a single execution schedule.

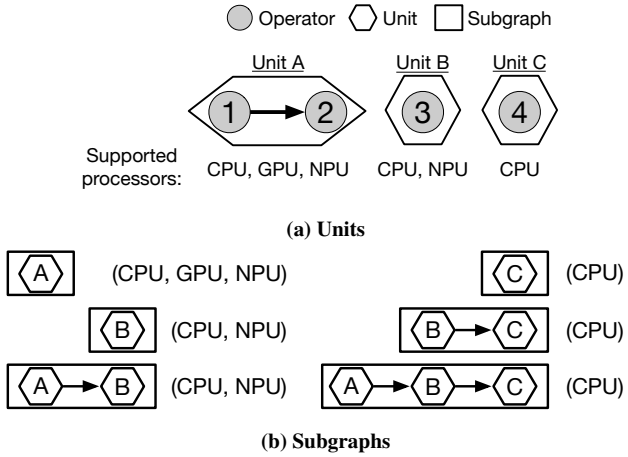
queue and picks an appropriate subgraph for the job, according to a pluggable scheduling policy, and schedules the subgraph on an appropriate processor. After the subgraph has finished running, if the subgraph covered the whole model, then the inference results are returned to the client. Otherwise, the job is enqueued back into the queue, with its execution progress marked correspondingly.

BAND spawns dedicated worker threads for each available processor. A worker thread runs at most one subgraph at a time on its processor. The number of worker threads for each processor is adjusted by a configurable system parameter.

4.2 Subgraph Partitioning

The model analyzer examines a registered model and creates specific subgraphs to be used by the scheduler. The analyzer aims to prepare a diverse set of subgraphs so that many possible schedules can be considered at runtime.

Take the model in Figure 9a as an example. The GPU only supports operators 1 and 2, while the NPU only supports operators 1, 2, and 3. Generally, all operators are always supported by the CPU. There are $3 \times 3 \times 2 \times 1 = 18$ possible methods of executing this



(a) Units

(b) Subgraphs

Figure 10: Grouping operators into units and subgraphs.

model, in terms of operator-processor placement. Existing frameworks [7, 8, 29] do not consider these various schedules, and only utilize one specific schedule[1]. Namely, TensorFlow Lite [8] groups adjacent operators based on whether an operator is supported on a processor or not. Operators that are supported on that processor are always run on that processor, while fallback operators are always run on the CPU, as shown in Figure 9b and 9c.

However, this does not necessarily imply that BAND should take all possible schedules into account. For instance, the schedule 1 (NPU) - 2 (GPU) - 3 (NPU) - 4 (CPU) is technically valid, but this involves a large number of transitions across processors, potentially resulting in excessive communication overheads. Rather than deriving schedules from the operator level to enumerate all possible schedules, BAND generates schedules from subgraphs. This inherently prevents schedules with frequent transitions from being selected. In a sense, BAND's partitioning method can be deemed as the middle ground between model-level scheduling (most coarse-grained) and operator-level scheduling (most fine-grained).

Units and subgraphs. Instead of directly creating subgraphs from operators, the model analyzer goes through two steps to create subgraphs. First, the model analyzer generates a list of *units* from the operators (Figure 10a). A unit is a group of adjacent operators that are all supported by the exact same set of processors. Each operator belongs to only one unit, i.e, no two units have a common operator. Also, units are created as big as possible. For instance, operators 1 and 2 form a unit that is supported by the CPU, GPU, and NPU. Operator 3 is made into a single-operator unit, as both neighboring operators (2 and 4) are supported by a different set of processors.

Next, the model analyzer generates subgraphs from these units (Figure 10b). Similar to how units were created, a subgraph is generated by grouping adjacent units but with different grouping conditions. The subgraph generation process starts by making a single-unit subgraph for each unit. Afterward, adjacent subgraphs can be merged into larger ones if they have at least one commonly supported processor. For example, the subgraph {1,2,3} is generated from {1,2}

[1]In the case of MNN [29] and Mace [7], operator adjacency is checked based on client-given operator indices (usually in the model file) rather than actual operator dependencies, resulting in overly fine-grained subgraphs.

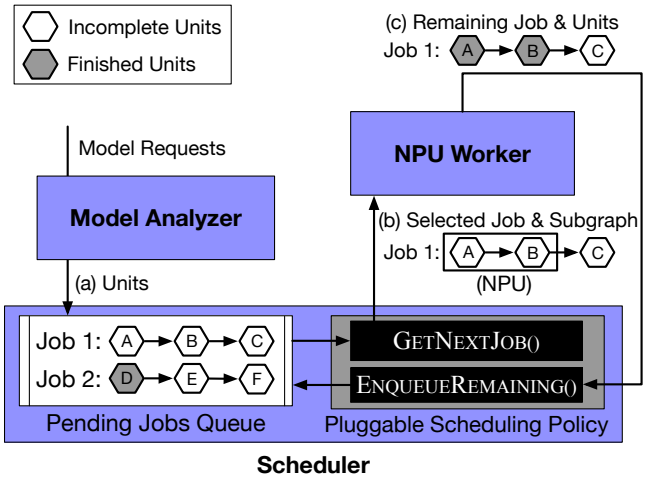


Figure 11: Detailed workflow of BAND's scheduler. (a) The scheduler spawns a job for each inference request, with units provided by the model analyzer. (b) The jobs are enqueued into the job queue, and the scheduling policy checks the queue to select the next job to process. The policy also chooses the subgraph and the processor to run. (c) Afterwards, the job's unit execution status is updated, and the job is put back into the job queue if there are any remaining units. The enqueue position of the updated job is determined by the policy.

and {3}. The set of supported processors for the {1,2,3} is the intersection of the processor sets of {1,2} and {3}. Note that the original small subgraphs are not discarded on merge.

Subgraph usage and memory. One interesting fact to note is that not all subgraphs are equally used, in terms of usage rates. Some subgraphs are severely underused, due to their execution times being too long. For instance, in Figure 10b, the subgraph {A,B,C} = {1,2,3,4} will most likely never be used, because running all units (i.e., all operators) on the CPU is generally too slow to satisfy tight SLO requirements. Such subgraphs needlessly occupy memory without benefiting the system. Therefore, it may be wise to not generate these subgraphs at all. Although BAND currently does not incorporate this optimization at the moment, this method provides a means of deleting scheduling candidates to save memory, which may be helpful for edge devices with severely limited memory.

4.3 Subgraph Scheduling

BAND's scheduler receives inference requests from client applications and schedules the requests to the worker threads according to an internal scheduling policy. Each inference request is translated into subgraph units, based on the analysis from the model analyzer. The policy examines the executable subgraphs from the units, and selects which subgraph to run. The scheduler also selects the processor to run that subgraph on.

4.3.1 Scheduling. In contrast to a fixed-processor policy as in TensorFlow Lite [8], MNN [29], and Mace [7], BAND's policies can utilize multiple processors at once to serve inference requests from multiple applications. The scheduling policy is a pluggable component that grants BAND flexibility in choosing the desired schedule

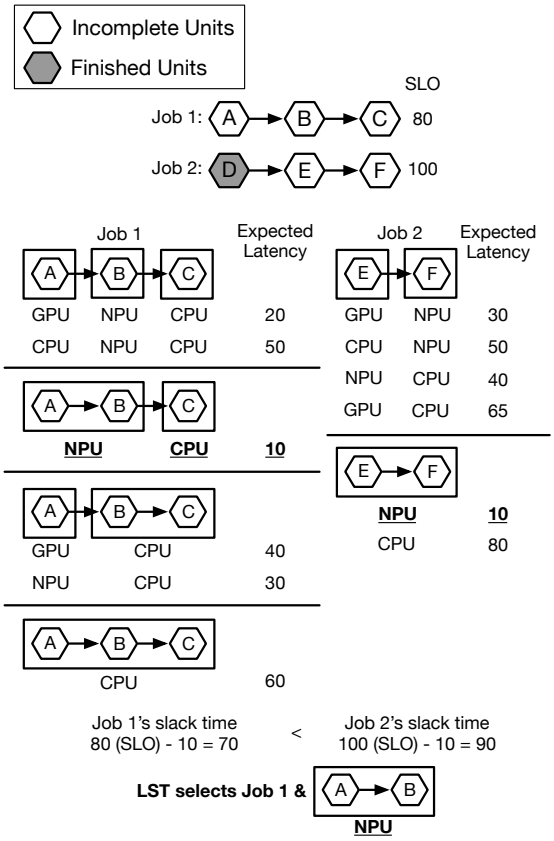


Figure 12: The Least Slack Time (LST) policy. The policy selects the job with the least slack, which is calculated by subtracting the expected latency from the SLO. The latency of the shortest subgraph sequence is regarded as that job's latency, and the first subgraph of that sequence is returned.

of a given workload (desired by that specific policy). Clients can configure which policy to use when the system starts up. Basic policies are implemented in BAND such as least slack time (LST), and clients can also implement their own custom policy.

Figure 11 depicts the overall scheduling flow of BAND. For each inference request, the scheduler spawns a job and tags it with the units of the requested model, looked up by consulting the model analyzer (Figure 11a). The spawned jobs are put in a separate queue.

Whenever a job is enqueued into the job queue, or a worker finishes executing a subgraph, the scheduling policy is notified. Specifically, the GETNEXTJOB() function is invoked, in which the policy selects the next job to be scheduled (Figure 11b). The job queue is not necessarily a FIFO queue; the scheduling policy is free to consider as many jobs as it wants, and is allowed to select a job that is not at the head of the job queue. For the chosen job, the policy must also pick a subgraph from the model analyzer (Section 4.2), and the processor to run the subgraph on. The scheduler then runs the subgraph on the chosen processor.

Selecting a job from the job queue. Each scheduling policy has its own unique logic for selecting a job and its subgraph. Figure 12 illustrates how BAND's default LST policy selects a job and subgraph, for a given set of jobs. The policy checks the slack time of each job

by comparing the job's SLO with its expected latency. Since the expected latency of a job differs based on which subgraphs are run for the job, the policy considers all valid subgraph sequences and picks the sequence with the shortest latency. A valid sequence is a permutation of subgraphs that covers each unit exactly once.

For example, for Job 1 in Figure 12, the subgraph sequence $\{A,B\}$-$\{C\}$ is valid, but the sequence $\{A,B\}$-$\{B,C\}$ is not because unit B would be run twice. Also, a single subgraph sequence may have multiple expected latencies because at least one of the subgraphs can be run on more than one processor, such as the sequence $\{A\}$-$\{B,C\}$. In that case, only the processors that lead to the least expected latency are considered ($\{A\}$ (NPU) -$\{B,C\}$ (CPU)). In the end, the job with the least slack (= SLO − latency), and the first subgraph of the subgraph sequence of that job is selected. Job 1 has a slack time of 70 (=80−10) because its SLO is 80 and the subgraph sequence $\{A,B\}$ (NPU) -$\{C\}$ (CPU) has an expected latency of 10. Job 2 has a slack time of 90 (=100−10) because its SLO is 100 and the subgraph sequence $\{E,F\}$ (NPU) has an expected latency of 10. Thus, Job 1 and the subgraph $\{A,B\}$ (NPU) is selected.

Note that the expected latency of a subgraph sequence is not simply equal to the sum of each individual subgraph's profiled execution time. Generally, subgraphs from other jobs are already being executed on processors at the time of scheduling, and thus the expected latency is usually larger than the sum of execution times. Thus, the scheduler must dynamically compute the expected latency every time GETNEXTJOB() is invoked. We found that a naive exhaustive search for all possible subgraph sequences incurs significant scheduling overhead at runtime – for n subgraphs with p compatible processors each, an exhaustive search would have a complexity of $O\left(p^n\right)$ (for a single job). Instead, the LST policy runs an algorithm based on dynamic programming to reduce the complexity to $O\left(pn^2\right)$. We observed that our scheduler takes no more than 0.1 ms to decide which subgraph to enqueue (Figure 11b), for our workloads.[2]

Scheduling the remaining units of a job. Note that the scheduling policy is not required to designate processors for all units of a job at once. The remaining units of a job that were not selected by the scheduling policy are enqueued back into the job queue (Figure 11c), after the selected subgraph finishes running. Later, they are given back to the policy for subsequent scheduling, together with other jobs in the job queue. Here, the policy has liberty in where to enqueue the remaining units, via the ENQUEUEREMAINING() function. For example, the LST policy always puts remaining units at the head of the job queue, in order to quickly check if the SLO of the original request can still be satisfied. On the other hand, a round-robin policy may want to enqueue such units at the tail of the queue.

Handling processors with the thermal shutdown. In case a processor becomes unavailable due to thermal throttling, we disable its worker and exclude it from scheduling. The subgraph that the worker was processing is enqueued back into the scheduler so that another worker can take over. BAND monitors system events (e.g., `ThermalStatusListener` in Android) and polls the disabled worker to notify the scheduler when the processor becomes online again.

[2]For the EagleEye scenario on Google Pixel 4, the average scheduling overhead was 0.049 ± 0.06 ms. The scheduler was mapped to the primary core (2.84 GHz Kryo 485) of the Pixel device.

4.3.2 Execution Time Profiles. In order for the scheduling policy to devise an appropriate schedule for a given set of jobs, the system must maintain an execution time profile for models. In Section 3, we discussed the difficulty of accurately predicting the execution time of a DNN on mobile platforms. Even if we were to establish an accurate profile for a DNN model within a certain workload, the profile may not be accurate for other workloads due to the inconsistent processor frequencies managed by DVFS. As such, BAND does not perform any offline intensive profiling for newly registered models. Rather, when a model is first registered to BAND, the system runs the model a few times to retrieve baseline execution time values, and estimates the execution times of the model's subgraphs based on the baseline execution time. Afterward, online adjustments are constantly made to reflect the current workload pattern.

When a new model is registered, BAND sequentially runs the largest subgraph of the model for each available processor and records the latencies. In order to avoid severely disrupting running jobs, BAND waits until the current job on a processor finishes, then pauses the corresponding worker so that the scheduler does not schedule subsequent jobs on that processor. After measuring the execution time of the subgraph on the processor during this pause, the scheduler is immediately notified that the worker is available again, and the scheduler proceeds as normal. This way, an unfortunate case where a job silently runs on a processor at the same time as profiling does not occur. Moreover, processors are paused one at a time, so jobs can still be scheduled to other processors; there is no need to halt the whole system for the sake of profiling.

After retrieving execution time values for the largest subgraphs per available processor, BAND measures the number of floating-point operations (FLOPs) of each subgraph, as well as the input and output tensors sizes. Assuming the execution time of a subgraph is roughly proportional to $FLOPs + \beta \times tensor\_size$, the execution time is estimated from the execution time of the largest subgraph in each processor. β is a constant that reflects the ratio between memory copy and computation; we used a value of $\frac{1{,}000}{bytes}$ for accelerators and $\frac{10}{bytes}$ for CPUs in our experiments, as accelerators are generally faster than CPUs.

Lastly, we adjust the latency profile of a subgraph at runtime with a simple linear smoothing function: $time_{profiled} \leftarrow \alpha\, time_{new} + (1 - \alpha)\, time_{profiled}$. α is a configurable system parameter between 0 and 1. We found that a value of 0.1 for α yields good performance, as we can prevent sudden runtime fluctuations from severely affecting latency profiles. Our method of predicting subgraph latencies, together with online latency profile adjustments, is sufficient for most of the models in our target workloads.

5 IMPLEMENTATION

We implemented BAND on top of TensorFlow Lite 2.3.0 [8], adding around 9.5K lines of C++ (core system) and Java (JNI for Android applications) code.

5.1 Android Service & Multi-App Support

BAND runs as an Android Service in the background. BAND defines APIs in AIDL [1] for client applications and implements the interfaces so as to instantiate Android Binder objects, which can be used by clients. Clients can fetch the Binder objects via `OnBind()` after

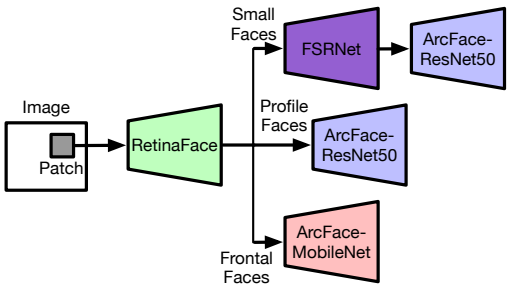


Figure 13: The EagleEye [57] workflow. After running the RetinaFace [18] face detection model on a patch of the input image, a different DNN is run on the faces of that same patch depending on the detected face types. The exact number of patches and faces are different for each frame.

binding with BAND, and call APIs through the binder to communicate with the system. Multiple clients can access BAND concurrently.

A client (an Android Activity or Service) can register DNN models to BAND via `modelId = registerModel(model)`. The client can then issue inference requests to the system, either in a synchronous manner by calling `output = runSync(modelId, input)`, or in an asynchronous manner by calling `reqId = runAsync(modelId, input)`. In the latter case, results can be retrieved via `output = wait(reqId)`.

When more than one application registers the same model to BAND, the system manages them as a single model to avoid resource duplication. Note that although the IPC overhead increases proportionally to the number of concurrent applications and parameter sizes of Binder function calls, the overheads (< 1ms) are relatively small compared to the typical model inference latency (> 10ms). Thus, the overheads did not significantly affect our experiments.

6 EVALUATION

In this section, we evaluate BAND's performance and examine the effects of its various techniques. BAND outperforms TensorFlow Lite by up to 5.04× for single-app workflows, where frames consist of dynamically set numbers of model inferences. Ablations studies demonstrate that BAND's subgraph mechanisms, profiling method, and thermal shutdown mechanism are effective. We also evaluate an environment where multiple apps execute each DNN task concurrently, to show how effectively BAND satisfies each app's SLO compared to the baselines.

6.1 Evaluation Setup

We execute various workloads on representative modern mobile devices with processors for DNN computation: Google Pixel 4 (Qualcomm Snapdragon 855 + Google Edge TPU), Xiaomi Redmi K40 Gaming (MediaTek Dimensity 1200), and Samsung Galaxy S20 (Qualcomm Snapdragon 865). We use TensorFlow Lite 2.3.0 as our baseline. Each experiment has its own specific baseline settings; they will be described in each section separately.

For all experiments, we create a single processor worker per accelerator to avoid contention. We add two single-threaded CPU workers mapped to BIG cores to parallelize CPU-only fallback operators.

6.2 Single-App: Back-to-Back Inference

We evaluate how well BAND performs on back-to-back inference workloads that have clearly defined "frames". A dynamically set number of inferences must occur for each frame before the next frame can be processed. We test the following workloads:

- **EagleEye [57]**: a person identification workflow (Figure 13) that runs face detection and identification models, depending on the number of faces in an image. The exact number of inferences per frame follows the YouTube Faces [53] dataset[3] (Figure 2). We randomly sampled 20% of faces to match the reported workload of EagleEye (∼ 20 faces per frame).
- **Distream [60]**: a live video analytics workload that detects objects in an image and applies attribute classification models, depending on the number of objects and type. The number of objects is determined from the ETH pedestrian [20] dataset[4].

We configure the baselines to create a TensorFlow Lite runtime instance for each model. Since the processor of a TensorFlow Lite runtime is fixed on instantiation, the model-processor mapping is also fixed. When an inference request for a model occurs, the baselines invoke the respective TensorFlow Lite runtime instance for execution. We set the model-processor mapping to minimize the makespan of inferences within a frame – the slowest model takes the fastest processor, etc. For instance, the models in the EagleEye workflow are distributed as: RetinaFace (CPU), ArcFace-MobileNet (GPU), ArcFace-ResNet50 (DSP), and FSRNet (NPU) for Google Pixel 4. We evaluate two types of baselines.

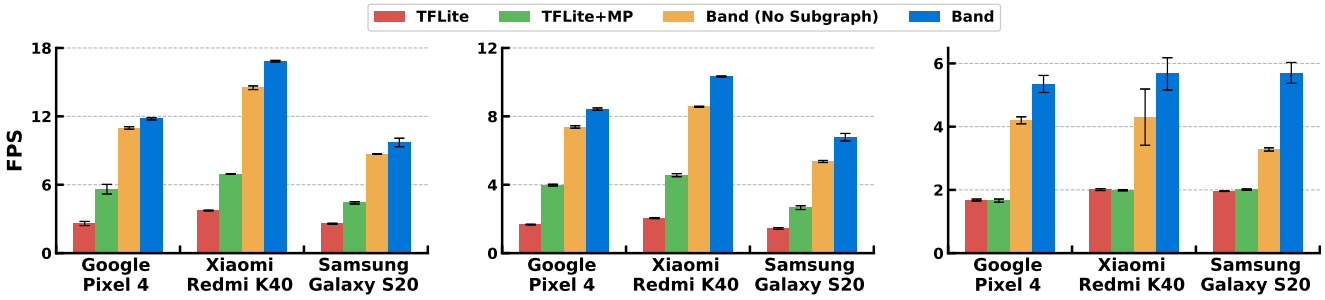
- **TensorFlow Lite (TFLite)**: Unmodified TensorFlow Lite 2.3.0. All operators (including non-fallback ones) after the first fallback operator of a model are run on the CPU, to minimize device transitions.
- **TFLite+MaxPartition (TFLite+MP)**: A slightly optimized version of TensorFlow Lite that continuously goes back and forth between the CPU and another processor, to run fallback operators and non-fallback ones, respectively.

We also show BAND's performance in two steps: **BAND**, our complete system, and **BAND (No Subgraph)**, which is based on our system but does not create subgraphs.

Figure 14a shows the results for the EagleEye workflow. BAND outperforms TFLite by up to 4.53×, and TFLite+MP by up to 2.42×. TFLite+MP is consistently faster than base TFLite, because TFLite+MP runs more operators on non-CPU accelerators. Our system is able to achieve even higher performance, by dynamically placing models on processors such that the makespan of a frame is minimized. Subgraph partitioning and scheduling also help, but the degree varies between devices because of the difference in operator coverage. We present more in-depth examinations on this matter in Section 7.3. Additionally, the Xiaomi Redmi K40 Gaming device has the highest overall performance of processors, hence the high FPS of K40 compared to the Pixel or S20.

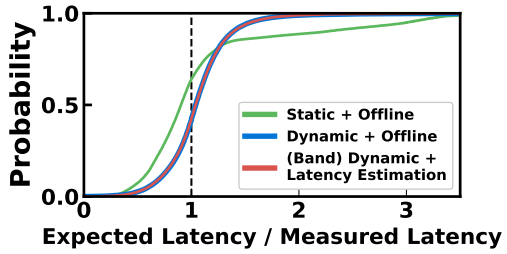
Figure 14b shows the results from the same EagleEye workflow, but for the top 10% crowded frames (i.e., frames with the most patches and faces). BAND outperforms TFLite by up to 5.04×, and

[3]The average number of model requests per frame is as follows; RetinaFace×6.0, FSRNet×1.5, ArcFace-MobileNet×5.2 and ArcFace-ResNet50×2.3.
[4]The average number of model requests per frame is as follows; EfficientDet-Lite2×5.0, EfficientNet-Lite1×5.6 and EfficientNet-Lite2×0.7.



(a) EagleEye (b) EagleEye (Crowded) (c) Distream

Figure 14: Processed FPS for varying workloads on various mobile devices.



(a) CDF of the ratio of expected to measured latency

	Static + Offline	Dynamic + Offline	BAND + Noise ± 30%	BAND
FPS	9.12 ± 0.11	9.46 ± 0.09	9.31 ± 0.13	9.46 ± 0.08
Profiling time (s)	76	76	4.8	4.8

(b) Processed FPS and profiling time-varying profiling methods

Figure 15: Effect of different profiling methods. FPS and latencies are measured by running the EagleEye workload 5 times with Samsung Galaxy S20.

TFLite+MP by up to 2.54× – the greater performance improvements come from the larger number of model inferences. BAND performs well when there is a sufficient number of inferences to coordinate.

Figure 14c reports experimental results for the Distream workflow. BAND outperforms TFLite by up to 3.18×, and TFLite+MP by up to 3.22×. Interestingly, TFLite+MP is not actually better than TFLite for the Distream workflow; the frequent device transitions outweigh the performance gains earned by utilizing non-CPU accelerators, resulting in worse FPS. This is a similar effect to having overly fine-grained subgraphs, as mentioned in Section 4.2. Regardless, BAND exhibits good performance on all devices. Furthermore, the gap between BAND and BAND (No Subgraph) is larger than that of the EagleEye workflow – especially for the S20. BAND enjoys performance gains from devising subgraph schedules derived from the fallback-abundant EfficientDet model present in Distream.

6.2.1 Analysis on Profiling. As stated in Section 4.3.2, BAND manages execution time profiles via initial latency estimations and dynamic online adjustments. This allows new models to be registered on the fly with a relatively quick initialization phase.

Figure 15a shows how accurate the execution time profiles are, for various profiling methods on the EagleEye workflow. The Static+Offline method measures the execution time of all subgraphs of a model before deploying the model, and does not further adjust the measured times at runtime. Compared to Band, the execution times measured offline are significantly shorter than the actual execution times, hence the warped CDF line. Meanwhile, the Dynamic+Offline method goes through the same offline step as Static+Offline, but also applies the runtime adjustments that BAND does. Both the Dynamic+Offline method and BAND demonstrate similar execution time profiles, but BAND has a much shortest initial profiling time (Figure 15b), making it less burdening to dynamically register a new model at runtime. Also, we observe that a ±30% noise to BAND's latency estimation does not significantly affect BAND's overall performance.

6.3 In-depth Analysis of Subgraphs

Scheduling with subgraphs gives us improved schedulability with more flexible, fine-grained schedules. We present a more detailed analysis of the benefits of scheduling models with subgraphs. To better understand how each model benefits from subgraphs, we conducted two case studies. First, we show latencies for frames consisting of the same model. Second, we report scheduling timelines, which reveal the scheduling decisions made by BAND.

6.3.1 Single Model Scalability. Figure 16 reports the frame latencies for four different models on Google Pixel 4, while changing the number of requests within a frame. Each frame is executed multiple times, and the average latency is reported on the graph.

In general, the subgraph-level execution of BAND outperforms the model-level execution BAND (No Subgraph). Compared to BAND (No Subgraph), BAND shows a latency speedup up to 1.40× (FSRNet). The latency improvement is from utilizing the preferred processors more efficiently, with the help of subgraph partitioning and scheduling. For instance, when fallback operators (e.g., TransConv) appear while executing FSRNet [15] on the NPU, the scheduler can yield the accelerator to the following requests so that the processor does not stay idle.

The RetinaFace [18] model in Figure 16d shows the least amount of speedup (1.11×) among the four models, when comparing BAND with BAND (No Subgraph). Even though the model can be executed with multiple subgraphs, the model latency on each processor does not vary by much, so there is little room for improvement with subgraph-level execution. In addition, since the positions of fallback

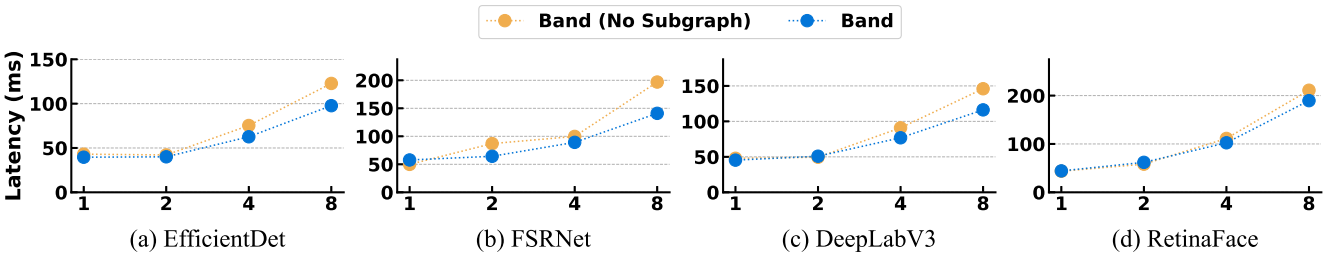
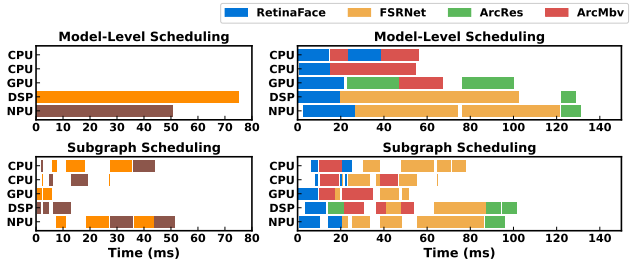


Figure 16: Frame latencies of running multiple instances of a model. For models with largely varying subgraph execution times such as EfficientDet and FSRNet, the gap between BAND and BAND (No Subgraph) is significant.



(a) Single frame with 2×FSRNet (b) Single frame of EagleEye workload

Figure 17: Subgraph scheduling timelines. Timelines where subgraph scheduling finds better schedules than model-level scheduling are shown. Both timelines are measured on Google Pixel 4.

operators are critical in deciding the model partition points, models with fewer fallback operators benefit less from subgraph scheduling.

6.3.2 Timeline Analysis. Figure 17a displays a case of running two FSRNet on Google Pixel 4. For model-level scheduling, DSP and NPU are allocated to the two requests since the two processors are faster than the other processors. However, it turns out that FSRNet was constantly going to the CPU to run fallback operators. Subgraph-level scheduling comes up with an alternate schedule that explicitly schedules these CPU fallback operators, successfully exploiting the idle times and reducing the makespan of the two models by 32%.

Figure 17b reports one of the scheduling decisions made by BAND while running the workload in Figure 14a. BAND considers fallback operators from RetinaFace and FSRNet to avoid any silent occupation on fast accelerators that leads to the under-utilization of the system. Compared to the model-level scheduling, ArcFace-MobileNet and ArcFace-ResNet extensively utilize fast accelerators such as DSP to leverage the benefit of subgraph-level scheduling. In this particular case, the frame latency is reduced by 23%.

6.4 Multi-App: Dynamic Input Workloads

As noted in Section 2, mobile devices can receive requests from various applications with their own requirements. Since each application is not aware of what processors the other applications are utilizing, multiple applications can happen to send requests to the same processor, leading to performance degradation.

In this section, we evaluate BAND with a scenario adapted from DeepEye [42]; a workload that runs four life-logging apps (scene [42], memory [32], age [39], and emotion [40]) that periodically request a single model inference with a sampling rate of 30 FPS. Each app

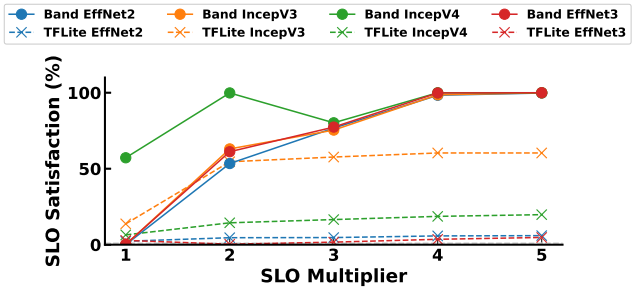


Figure 18: SLO satisfaction ratio of running apps on Pixel 4.

	TFLite+MP	BAND
Power (W)	7.60	**7.99**
FPS	4.11	**8.71**

Table 2: Average power consumption in Google Pixel 4 while processing the EagleEye (Crowded) workflow. Power consumption was measured using a Monsoon Power Monitor.

is backed by distinct classification backbones such as EfficientNet-Lite2, EfficientNet-Lite3, InceptionV3[50], and InceptionV4[49], respectively. For each model, we use SLO requirements of the 99th percentile of isolated execution time on the fastest processor times multiplier. We use the *Least Slack Time* policy and compare BAND with a baseline having static model-processor mappings where each app sends model requests to the fastest processor.

Figure 18 shows the average SLO violation rates of 5 experiments for varying SLO requirements on Google Pixel 4. BAND outperforms TFLite by up to 3.76× on average in a tight deadline of 99th percentile×2. With TFLite, latency-critical requests easily miss the required SLO due to having no coordination on the processor contention of NPU. In contrast, BAND coordinates heterogeneous processors to achieve a higher satisfaction rate. In case of tight deadlines (Multiplier = 1, 2), BAND leverages the relatively high slack time of InceptionV4 to minimize the violation.

6.5 Power Consumption

Finally, we report the power consumption of BAND in Table 2. We obtained power measurements from a Monsoon Power Monitor [12] attached to a Google Pixel 4 device. For the EagleEye (Crowded) workload, BAND consumes 7.99W on average (measured during no throttling). Considering the base power consumption of TFLite+MP, BAND uses 5.1% more power to process 2.1× frames.

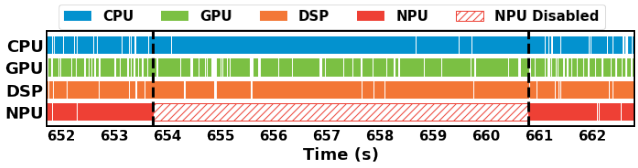


Figure 19: EagleEye scheduling timeline on Google Pixel 4 before and after the NPU becomes unavailable due to throttling.

Both BAND and TFLite+MP consume more power than the 5W TDP [6] of the Pixel 4, and thus thermal throttling may occur when the workload is run for an extended period of time. Figure 19 shows an actual case of throttling that occurred during our experiments, after running the EagleEye workload for more than 10 minutes.

The NPU is disabled for about 7 seconds, and BAND excludes the NPU from its candidates of processors when scheduling requests at this time (the thermal shutdown mechanism described in Section 4.3). During this period, the performance drops to 5.9 FPS, which is lower than BAND's ordinary performance but is still higher than TFLite+MP using all processors, including the NPU. After the NPU recovers, BAND detects that the NPU is usable again and starts to utilize it.

In contrast to BAND's post-throttling mechanism of excluding certain processors, a pre-throttling mechanism that detects such excessive power consumption beforehand may be more effective for maintaining a long-running service. We leave this as future work.

7 RELATED WORK

We describe related work relevant to BAND, categorized by their approach in serving multi-DNN workloads on multiple processors.

Multiple DNNs on a single processor. A handful of research efforts, including DeepEye [42], MCDNN ,[26] and NestDNN [21], run multiple DNNs on a single processor by applying model compression to change DNN structures, or assuming that computation can be shared among DNNs. In contrast, BAND focuses on coordinating a set of models given from apps without time-consuming model-based optimizations.

Layerweaver [43] concurrently runs compute-intensive operations (convolutions) of one DNN with memory-intensive operations (load weights) of another. Layerweaver exploits the fact that computations on NPU processing element (PE) arrays and memory accesses on the DRAM can occur at the same time, to achieve high resource utilization. Our work, on the other hand, does not impose any specific resource usage requirements on the DNNs. For instance, all DNNs used in the Distream workflow fall into the compute-intensive category of Layerweaver, so Layerweaver is unable to efficiently overlap DNN operations in Distream.

Masa [17] is similar to Layerweaver in that it tries to overlap compute operations with memory loads, but focuses more on the memory aspect, devising a memory-aware scheduler for devices with limited memory, up to 1 GB. BAND targets devices with much more memory (6-12 GB), so the performance boost gained from such a scheduler would be marginal – also shown in the Masa paper.

DART [54] is a scheduling framework that provides scheduling of real-time tasks and best-effort tasks, on heterogeneous platforms of CPUs and GPUs. DART's goal of coordinating multiple DNNs on heterogeneous processors aligns with BAND's goal, but specific hardware assumptions such as task preemption and shared CUDA contexts (NVIDIA GPUs) make the scheduler designs vastly different. Especially, there is no need to partition a DNN into smaller scheduling units (i.e., subgraphs) for DART, as a DNN can always be preempted or run concurrently with another DNN.

Single DNN on multiple processors. Several works propose methods for accelerating a single DNN on multiple mobile processors [33, 52]. µLayer [33] partitions a DNN layer into channels (for conv and FC) and distributes the computation of channels across the CPU and GPU. Similarly, AsyMo [52] divides a matrix multiplication operation into subtasks in order to utilize heterogeneous CPU cores. In order to incorporate these techniques, BAND's subgraph-centric coordination model can be extended to run a subgraph on multiple workers, rather than only a single worker at a time.

Multiple DNNs on multiple processors. Application-based approaches [21, 22, 38, 41, 42, 57] suggest coordinating asymmetric computing resources to process computation pipelines for tasks, without considering interference. LEO [22] and Starfish [41] proposed a centralized framework for computation and memory sharing between similar tasks of sensing and continuous vision applications. MobiSR [38] and EagleEye [57] adopt content-aware optimizations for efficient processing of tasks. BAND is complementary to these approaches, as it is agnostic to such content-based optimizations.

Recent works have recognized that fine-grained execution of DNN models can increase the schedulability of non-preemptive heterogeneous processors [25, 48, 58]. MOSAIC [25] automatically generates an optimal partitioning plan for accurate and efficient inference, but is limited to a single DNN and manipulates processor frequencies for runtime predictability. Seo *et al.*[48] and Heimdall [58] evenly distribute the total inference time across multiple partitions to achieve specific scheduling goals such as SLO-awareness or mitigating GPU contention between rendering and DNN inference. All efforts to date utilize profiling results collected offline, but the varying runtime performance of processors would significantly affect their scheduling objectives due to incorrect latency predictions. BAND is robust to such mobile processor performance dynamics, and further considers merging subgraphs to mitigate the overhead of fine-grained execution.

8 CONCLUSION

In this paper, we presented BAND, a mobile inference system that coordinates multi-DNN workloads on heterogeneous processors. BAND builds scheduling units based on operator supportability on each processor, and schedules multiple requests with pluggable policies according to the application requirements. For single application scenarios, BAND provides up to 5.04× improved FPS compared to TensorFlow Lite. For a multi-application scenario, BAND demonstrates up to 3.76× better SLO satisfaction rates.

ACKNOWLEDGEMENT

We sincerely thank our shepherd Matt Welsh and the anonymous reviewers for their valuable comments. This work was supported by Samsung Research Funding & Incubation Center of Samsung Electronics under project number SRFC-IT2001-03.

REFERENCES

[1] 2008. Android Interface Definition Language (AIDL). https://developer.android.com/guide/components/aidl.
[2] 2008. NVIDIA CUDA. https://developer.nvidia.com/cuda-toolkit.
[3] 2017. Google NNAPI. https://developer.android.com/ndk/guides/neuralnetworks.
[4] 2017. Tencent NCNN. https://github.com/Tencent/ncnn.
[5] 2018. Google EdgeTPU. https://cloud.google.com/edge-tpu.
[6] 2018. Qualcomm Snapdragon 855 Specs. https://www.notebookcheck.net/Qualcomm-Snapdragon-855-SoC-Benchmarks-and-Specs.375436.0.html.
[7] 2018. Xiaomi MACE. https://github.com/XiaoMi/mace.
[8] 2019. TensorFlow Lite. https://www.tensorflow.org/lite.
[9] 2019. Using deep neural networks for accurate hand-tracking on Oculus Quest. https://ai.facebook.com/blog/hand-tracking-deep-neural-networks/.
[10] 2020. Huawei HiAi DDK. https://developer.huawei.com/consumer/en/hiai.
[11] 2020. Qualcomm Hexagon. https://developer.qualcomm.com/software/hexagon-dsp-sdk/dsp-processor.
[12] 2021. Monsoon High Voltage Power Monitor. https://www.msoon.com/high-voltage-power-monitor.
[13] Jonas Blattgerste, Benjamin Strenge, Patrick Renner, Thies Pfeiffer, and Kai Essig. 2017. Comparing Conventional and Augmented Reality Instructions for Manual Assembly Tasks. In *PETRA*.
[14] Michael Buch, Zahra Azad, Ajay Joshi, and Vijay Janapa Reddi. 2021. AI Tax in Mobile SoCs: End-to-end Performance Analysis of Machine Learning in Smartphones. In *ISPASS*.
[15] Yu Chen, Ying Tai, Xiaoming Liu, Chunhua Shen, and Jian Yang. 2018. Fsrnet: End-to-end learning face super-resolution with facial priors. In *CVPR*.
[16] Yujeong Choi and Minsoo Rhu. 2020. PREMA: A Predictive Multi-task Scheduling Algorithm For Preemptible Neural Processing Units. In *HPCA*. IEEE, 220–233.
[17] Bart Cox, Jeroen Galjaard, Amirmasoud Ghiassi, Robert Birke, and Lydia Y Chen. 2021. Masa: Responsive Multi-Dnn Inference on the Edge. In *PerCom*.
[18] Jiankang Deng, Jia Guo, Evangelos Ververas, Irene Kotsia, and Stefanos Zafeiriou. 2020. RetinaFace: Single-Shot Multi-Level Face Localisation in the Wild. In *CVPR*.
[19] Kuntai Du, Ahsan Pervaiz, Xin Yuan, Aakanksha Chowdhery, Qizheng Zhang, Henry Hoffmann, and Junchen Jiang. 2020. Server-Driven Video Streaming for Deep Learning Inference. In *SIGCOMM*.
[20] A. Ess, B. Leibe, K. Schindler, and L. van Gool. 2008. A Mobile Vision System for Robust Multi-Person Tracking. In *CVPR*.
[21] Biyi Fang, Xiao Zeng, and Mi Zhang. 2018. NestDNN: Resource-Aware Multi-Tenant On-Device Deep Learning for Continuous Mobile Vision. In *MobiCom*.
[22] Petko Georgiev, Nicholas D. Lane, Kiran K. Rachuri, and Cecilia Mascolo. 2016. LEO: Scheduling Sensor Inference Algorithms across Heterogeneous Mobile Processors and Network Resources. In *MobiCom*.
[23] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. In *OSDI*. 443–462.
[24] Myeonggyun Han and Woongki Baek. 2021. HERTI: A Reinforcement Learning-Augmented System for Efficient Real-Time Inference on Heterogeneous Embedded Systems. In *PACT*.
[25] Myeonggyun Han, Jihoon Hyun, Seongbeom Park, Jinsu Park, and Woongki Baek. 2019. MOSAIC: Heterogeneity-, Communication-, and Constraint-Aware Model Slicing and Execution for Accurate and Efficient Inference. In *PACT*.
[26] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. 2016. MCDNN: An Approximation-Based Execution Framework for Deep Stream Processing Under Resource Constraints. In *MobiSys*.
[27] Sinh Huynh, Rajesh Krishna Balan, JeongGil Ko, and Youngki Lee. 2019. VitaMon: Measuring Heart Rate Variability Using Smartphone Front Camera. In *SenSys*.
[28] Jun-Woo Jang, Sehwan Lee, Dongyoung Kim, Hyunsun Park, Ali Shafiee Ardestani, Yeongjae Choi, Channoh Kim, Yoojin Kim, Hyeongseok Yu, Hamzah Abdel-Aziz, Jun-Seok Park, Heonsoo Lee, Dongwoo Lee, Myeong Woo Kim, Hanwoong Jung, Heewoo Nam, Dongguen Lim, Seungwon Lee, Joon-Ho Song, Suknam Kwon, Joseph Hassoun, SukHwan Lim, and Changkyu Choi. 2021. Sparsity-Aware and Re-configurable NPU Architecture for Samsung Flagship Mobile SoC. In *ISCA*. 15–28.
[29] Xiaotang Jiang, Huan Wang, Yiliu Chen, Ziqi Wu, Lichuan Wang, Bin Zou, Yafeng Yang, Zongyang Cui, Yu Cai, Tianhang Yu, Chengfei Lv, and Zhihua Wu. 2020. MNN: A Universal and Efficient Inference Engine. In *MLSys*.
[30] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. 2017. Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge. In *ASPLOS*.
[31] David Kanter. 2015. Graphics processing requirements for enabling immersive vr. *AMD White Paper* (2015).
[32] Aditya Khosla, Akhil S Raju, Antonio Torralba, and Aude Oliva. 2015. Understanding and predicting image memorability at a large scale. In *ICCV*. 2390–2398.
[33] Youngsok Kim, Joonsung Kim, Dongju Chae, Daehyun Kim, and Jangwoo Kim. 2019. µLayer: Low Latency On-Device Inference Using Cooperative Single-Layer Acceleration and Processor-Friendly Quantization. In *EuroSys*.
[34] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. 2020. Do OS abstractions make sense on FPGAs?. In *OSDI*. 991–1010.
[35] Nicholas D. Lane and Petko Georgiev. 2015. Can Deep Learning Revolutionize Mobile Sensing?. In *HotMobile*.
[36] Nicholas D. Lane, Petko Georgiev, and Lorena Qendro. 2015. DeepEar: Robust Smartphone Audio Sensing in Unconstrained Acoustic Environments Using Deep Learning. In *UbiComp*.
[37] Stefanos Laskaridis, Stylianos I. Venieris, Mario Almeida, Ilias Leontiadis, and Nicholas D. Lane. 2020. SPINN: Synergistic Progressive Inference of Neural Networks over Device and Cloud. In *MobiCom*.
[38] Royson Lee, Stylianos I. Venieris, Lukasz Dudziak, Sourav Bhattacharya, and Nicholas D. Lane. 2019. MobiSR: Efficient On-Device Super-Resolution through Heterogeneous Mobile Processors. In *MobiCom*.
[39] Gil Levi and Tal Hassner. 2015. Age and gender classification using convolutional neural networks. In *CVPRW*. 34–42.
[40] Gil Levi and Tal Hassner. 2015. Emotion recognition in the wild via convolutional neural networks and mapped binary patterns. In *ICMI*. 503–510.
[41] Robert LiKamWa and Lin Zhong. 2015. Starfish: Efficient Concurrency Support for Computer Vision Applications. In *MobiSys*.
[42] Akhil Mathur, Nicholas D. Lane, Sourav Bhattacharya, Aidan Boran, Claudio Forlivesi, and Fahim Kawsar. 2017. DeepEye: Resource Efficient Local Execution of Multiple Deep Vision Models Using Wearable Commodity Hardware. In *MobiSys*.
[43] Young H Oh, Seonghak Kim, Yunho Jin, Sam Son, Jonghyun Bae, Jongsung Lee, Yeonhong Park, Dong Uk Kim, Tae Jun Ham, and Jae W Lee. 2021. Layerweaver: Maximizing Resource Utilization of Neural Processing Units via Layer-Wise Scheduling. In *HPCA*. 584–597.
[44] S. K. Park, M. S. O'Neill, P. S. Vokonas, D. Sparrow, and J. Schwartz. 2005. Effects of air pollution on heart rate variability: the VA normative aging study. *Environ Health Perspect* (2005).
[45] Hang Qiu, Fawad Ahmad, Fan Bai, Marco Gruteser, and Ramesh Govindan. 2018. AVR: Augmented vehicular reality. In *MobiSys*. 81–95.
[46] Kiran K. Rachuri, Mirco Musolesi, Cecilia Mascolo, Peter J. Rentfrow, Chris Longworth, and Andrius Aucinas. 2010. EmotionSense: A Mobile Phones Based Adaptive Platform for Experimental Social Psychology Research. In *UbiComp*.
[47] Bhargava Reddy, Ye-Hoon Kim, Sojung Yun, Chanwon Seo, and Junik Jang. 2017. Real-Time Driver Drowsiness Detection for Embedded System Using Model Compression of Deep Neural Networks. In *CVPRW*.
[48] Wonik Seo, Sanghoon Cha, Yeonjae Kim, Jaehyuk Huh, and Jongse Park. 2021. SLO-Aware Inference Scheduler for Heterogeneous Processors in Edge Platforms. *ACM Trans. Archit. Code Optim.* (2021).
[49] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. 2017. Inception-v4, inception-resnet and the impact of residual connections on learning. In *AAAI*.
[50] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *CVPR*. 2818–2826.
[51] Anuj Vaishnav, Khoa Dang Pham, and Dirk Koch. 2018. A survey on FPGA virtualization. In *FPL*. 131–1317.
[52] Manni Wang, Shaohua Ding, Ting Cao, Yunxin Liu, and Fengyuan Xu. 2021. AsyMo: Scalable and Efficient Deep-Learning Inference on Asymmetric Mobile CPUs. In *MobiCom*.
[53] Lior Wolf, Tal Hassner, and Itay Maoz. 2011. Face recognition in unconstrained videos with matched background similarity. In *CVPR*. 529–534.
[54] Yecheng Xiang and Hyoseung Kim. 2019. Pipelined data-parallel CPU/GPU scheduling for multi-DNN real-time inference. In *2019 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 392–405.
[55] Mengwei Xu, Zhe Fu, Xiao Ma, Li Zhang, Yanan Li, Feng Qian, Shangguang Wang, Ke Li, Jingyu Yang, and Xuanzhe Liu. 2021. From Cloud to Edge: A First Look at Public Edge Platforms. In *ACM IMC*.
[56] Amir Yazdanbakhsh, Kiran Seshadri, Berkin Akin, James Laudon, and Ravi Narayanaswami. 2021. An evaluation of edge tpu accelerators for convolutional neural networks. *arXiv preprint arXiv:2102.10423* (2021).
[57] Juheon Yi, Sunghyun Choi, and Youngki Lee. 2020. EagleEye: Wearable Camera-Based Person Identification in Crowded Urban Spaces. In *MobiCom*.
[58] Juheon Yi and Youngki Lee. 2020. Heimdall: Mobile GPU Coordination Platform for Augmented Reality Applications. In *MobiCom*.
[59] Jiadi Yu, Hongzi Zhu, Haofu Han, Yingying Jennifer Chen, Jie Yang, Yanmin Zhu, Zhongyang Chen, Guangtao Xue, and Minglu Li. 2016. SenSpeed: Sensing Driving Conditions to Estimate Vehicle Speed in Urban Environments. *IEEE Transactions on Mobile Computing* (2016).
[60] Xiao Zeng, Biyi Fang, Haichen Shen, and Mi Zhang. 2020. Distream: scaling live video analytics with workload-adaptive distributed edge intelligence. In *SenSys*. 409–421.