# AI Tax in Mobile SoCs: End-to-end Performance Analysis of Machine Learning in Smartphones

Michael Buch[1], Zahra Azad[2], Ajay Joshi[2], and Vijay Janapa Reddi[1]

[1]Harvard University
[2]Boston University
{*mbuch, vjreddi*}*@g.harvard.edu,* {*zazad, joshi*}*@bu.edu*

***Abstract*—Mobile software is becoming increasingly feature rich, commonly being accessorized with the powerful decision making capabilities of machine learning (ML). To keep up with the consequently higher power and performance demands, system and hardware architects add specialized hardware units onto their system-on-chips (SoCs) coupled with frameworks to delegate compute optimally. While these SoC innovations are rapidly improving ML model performance and power efficiency, auxiliary data processing and supporting infrastructure to enable ML model execution can substantially alter the performance profile of a system. This work posits the existence of an *AI tax*, the time spent on non-model execution tasks. We characterize the execution pipeline of open source ML benchmarks and Android applications in terms of AI tax and discuss where performance bottlenecks may unexpectedly arise.**

***Index Terms*—mobile systems, Android, system-on-chip, hardware acceleration, machine learning, workload characterization.**

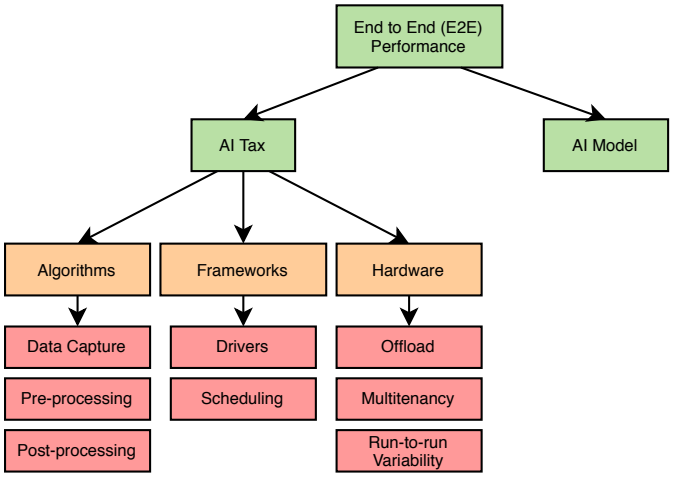



Fig. 1: Taxonomy of overheads for AI processing, which in this paper we expand upon to highlight common pitfalls when assessing AI performance on mobile chipsets.

## I. Introduction

With the advances in the compute and storage capacity of systems, there has been a surge of artificial intelligence (AI) in mobile applications. Many of today's mobile applications use AI-driven components like super resolution [1], face recognition [2], image segmentation [3], virtual assistants [4], speech transcription [5], sentiment analysis [6], gesture recognition [7], etc. To make an application accessible to the broader audience, AI needs to be routinely deployed on mobile systems, which presents the following three key challenges:

*First*, AI processing on general-purpose mobile processors is inefficient in terms of energy and power, and so we need to use custom hardware accelerators. Commonly used AI hardware accelerators include traditional GPUs and DSPs as well as custom AI processing engines such as Google's Coral [8], MediaTek's Dimensity [9], Samsung's Neural Processing Unit [10], and Qualcomm's AIP [11]. While these AI engines offer hardware acceleration, depending on how they are integrated into the wider system, they can potentially introduce offloading overheads that are often not accounted for or studied in AI system performance analyses.

*Second*, to manage the underlying AI hardware, SoC vendors rely on AI software frameworks. Examples include Google's Android Neural Network API (NNAPI) [12], Qualcomm's Snapdragon Neural Processing Engine (SNPE) [13], and MediaTek's NeuroPilot [14]. These frameworks manage ML model execution on heterogeneous hardware and decide on the application run-time scheduling plan. While this reduces development effort, abstracting model preparation and offload can make performance predictability difficult to reason about as the framework's run-time behavior varies depending on system configuration, hardware support, model details, and the end-to-end application pipeline.

*Third*, to enable the execution of a machine learning (ML) model, an application typically requires a host of additional algorithmic processing. Examples of such algorithmic processing include retrieval of data from sensors, pre-processing of the inputs, initialization of ML frameworks, and post-processing of results. While every ML application and its model(s) must go through these necessary processing stages, such overheads are often discounted from performance analysis.

In this paper, we explain the individual stages of execution of open-source ML benchmarks and Android applications on state-of-the-art mobile SoC chipsets [15]–[18] to quantify the performance penalties paid in each stage. We call the combined end-to-end latency of the ML execution pipeline the **"AI tax"**. Figure 1 shows the high-level breakdown

of an application's end-to-end performance and how end-to-end performance can be broken down into AI tax and AI model execution time. The AI tax component has three categories that introduce overheads, which include *algorithms*, *frameworks* and *hardware*. Each of these categories has its own sources of overhead. For instance, algorithmic processing involves the runtime overhead associated with capturing the data, pre-processing it for the neural network, and running post-processing code that often generates the end result. Frameworks have driver code that coordinates scheduling and optimizations. Finally, the CPU, GPU or accelerator itself can suffer from offloading costs, run-to-run variability and lag due to multi-tenancy (or running concurrent models together).

We use AI tax to point out the potential algorithmic bottlenecks and pitfalls of delegation frameworks that typical inference-only workload analysis on hardware would not convey. For example, a crucial shortcoming of several existing works, including industry-standard benchmarks such as MLPerf [19] and AI Benchmark [20], is that they overemphasize ML inference performance (i.e., AI Model in Figure 1). In doing so, they ignore the rest of the processing pipeline overheads (i.e., AI tax portion in Figure 1) — thereby "missing the forest for the trees". ML inference-only workload performance analysis commonly does not capture the non-model execution overheads because they are either interested in pure inference performance or the ad-hoc nature of these overheads makes it unclear how to model them in a standardized fashion. However, since the high-level application execution pipeline varies marginally between ML models and frameworks, many ML mobile applications follow a similar structure, and so patterns of where bottlenecks form could thus be captured systematically and that could help us mitigate inefficiencies.

To that end, we characterize the importance of end-to-end analysis of the execution pipeline of real ML workloads to find the contribution of AI tax in the machine learning application time. We examine this from the perspective of algorithmic processing, framework inefficiencies and overheads, and hardware capabilities. Given that data capture and processing times can be a significant portion of end-to-end AI application performance, these overheads are essential to quantify when assessing mobile AI performance. Moreover, it is necessary to consider jointly accelerating these seemingly mundane yet important data processing tasks along with ML execution. We also present an in-depth study of ML offload frameworks and demonstrate how they can be effectively coupled with a measure of AI tax to aid with efficient hardware delegation decisions. Finally, application performance can vary substantially based on the underlying hardware support, but this is not always transparent to the end user. Hence, this is yet another reason that end-to-end AI application performance (i.e., including the software framework) is crucial instead of just measuring hardware performance in isolation.

In summary, our contributions are as follows:

1) It is important to consider end-to-end performance analysis when quantifying AI performance in mobile chipsets. Specifically, the current state-of-the-art benchmarks overly focus on model execution and miss the system-level implications crucial for AI usability.
2) The data capture and the AI pre- and post-processing stages are just as crucial for AI performance as the core ML computational kernels because they can consume as much as 50% of the actual execution time.
3) When conducting AI performance analysis, it is essential to consider how well the ML software framework is optimized for the underlying chipset. Not all frameworks support all chipsets and models well.
4) AI hardware performance is susceptible to cold start penalties, run-to-run variation, and multi-tenancy overheads, much of which are all overlooked today.

We believe that this breakdown of end-to-end AI performance is of both interest and significance for users across the stack. End-users (i.e., application developers) care because they may want to invest in additional complexity to lower pre-processing overheads (instead of focusing on inference). Framework authors may wish to report more details about pre- and post-processing or data collection to its users. Framework writers may also want to think about scheduling pre- and post-processing operations on hardware. AI accelerator designers may want to consider dropping an expensive tensor accelerator in favor of a cheaper DSP that can also do pre-processing.

The remainder of the paper is organized as follows. Section II describes the steps involved in using an ML model from within a mobile application, from data acquisition to interpretation of the results. In Section III, we detail our experimental setup, including the frameworks, models, applications, and hardware that we tested. We define AI tax and quantify it in Section IV. We discuss prior art in Section V and conclude the paper in Section VI.

## II. The Machine Learning Pipeline

This section presents the typical flow of events when invoking an ML model on an SoC, which we refer to as the *ML pipeline*. This sets the stage for how we conduct our end-to-end AI performance analysis. Figure 2 shows each of the stages, and the section elaborates on their individual functions.

### A. Data Capture

Acquiring data from sensors can seem trivial on the surface, but can easily complicate an application's architecture and influence system design decisions. For example, capturing raw images faster than what the application can handle can put strains on system memory and I/O or an incorrect choice of image resolution can cause non-linear performance drops if image processing algorithms in later parts of the ML pipeline do not scale with image size. Some systems collect data from more than a single sensor, in which case additional data processing (such as fusing multiple sources of data into a single metric) or sanitization may be required. This extra processing often runs concurrently with the rest of the application across the same set of cores adding synchronization overheads and interference into the picture. Alternatively it may also be offloaded to co-processors, such as the DSP

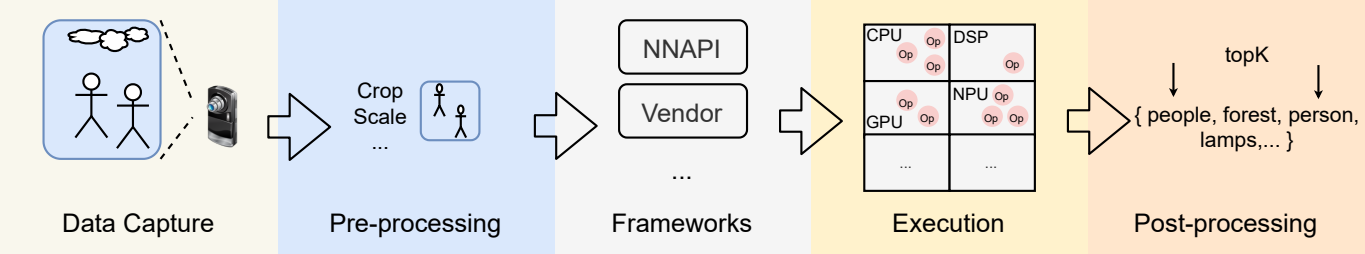



Fig. 2: Example of the steps taken in a typical end-to-end flow for an image classification machine learning task. A *pre-processing* step transforms *data captured* by the camera into the shape that the model in question expects. The *ML framework* loads the model and plans any subsequent stages; the choice of framework can be platform agnostic (e.g., NNAPI) or provided by a chipset vendor (e.g., Qualcomm's SNPE) and its initialization is typically performed once. Then the framework schedules *model execution* on hardware at "operation" granularity. Finally, a *post-processing* step makes sense of the model output, e.g., by selecting the likeliest classes to which the image belongs (i.e., topK).

through frameworks such as Qualcomm's FastCV [21], which bring additional performance considerations of their own. In the applications we studied, the supporting code around data capture contributed to a large share of overall application latency.

### B. Pre-processing

The first step in using sensor data is to shape it into a format that the model in question expects. Although ML frameworks expose some standard pre-processing algorithms, the input requirements of a model depend on the task, dataset, network architecture, and application details, such as how data is acquired and stored. Hence, a system's architecture can affect the choice of pre-processing algorithms and the time spent in that stage. In this paper, we deal with models that operate on image data, and so we direct our attention in the remainder of this section to the common algorithms used for image pre-processing in the applications we benchmarked:

**Bitmap formatting**: When dealing with image data that we obtain from a camera module, an application needs to convert the raw frames into some standard image representation. A common pattern in Tensorflow-based Android applications is to retrieve a camera frame in the YUV NV21 format using the Android Camera API [22] and convert it to ARGB8888 format.

**Scale/Crop**: A model is trained on images of fixed dimensions, and the input dimensions determine a network's architecture (i.e., the sizes of individual layers). Hence, somewhere in the pre-processing stage, the image data needs to get resized to the size the first layer of the network can handle. A widespread technique for scaling images (e.g., used as Tensorflow's default resizing algorithm) is *Bilinear Interpolation*. Although this algorithm's run-time scales quadratically with the output image size, vectorized implementations can significantly improve its performance. Some models such as Inception-v3 (center-)crop an image prior to scaling it [23]. This step removes some number of pixels along each dimension of an image. This operation's cost is thus the cost of computing the bounding box around the cropped region and reshaping a tensor. In the majority of models we measured, cropping and scaling was a significant contributor to overall execution time.

**Normalize**: Almost all networks require normalized inputs, i.e., with zero mean and unit variance. A typical implementation will thus iterate over each pixel, subtract the image mean from it and scale it by the standard deviation. The run-time of a normalization operation thus scales linearly with the number of pixels in the input image.

**Rotate**: Depending on the orientation of the image provided by the sensors, an application might have to perform an additional rotation operation on the input. If a network was trained on augmented data or if the sensor orientation is consistent with the image orientation expected by the model, this step might not be necessary. However, models such as PoseNet [24], which we evaluate in this paper, make extensive use of this operation, and it may be essential to keep in mind that image rotation scales quadratically with the image size.

**Type conversion**. Quantized and reduced precision models (such as 16-bit floating-point) are increasingly popular, especially in mobile systems because less memory is required to store weights and activations while accuracy remains in acceptable bounds. However, the datatype and bit-width of a model also affects the pre-processing stage because frames from sensors are captured as raw bytes and thus need to be converted to the appropriate types and possibly be quantized.

### C. Frameworks

Most of the ML pipeline is determined by the framework(s). Pre- and post-processing algorithms are often provided as library calls, such as TensorFlow operations in TFLite. Input data formats, model execution, and computation offload is all managed transparently (and often configurable manually) by a run-time, e.g., Android's NNAPI. Device dependent frameworks are offered by most SoC vendors, e.g., Qualcomm's SNPE or MediaTek's ANN, but there is ongoing effort to provide interoperability between them and NNAPI. They can expand the list of accelerators on which we can schedule computation and a model's supported operations. Given that

| Task | Model | Resolution | Pre-processing Task | Post-processing Task | NNAPI-fp32 | NNAPI-int8 | CPU-fp32 | CPU-int8 |
|---|---|---|---|---|---|---|---|---|
| Classification | MobileNet 1.0 v1 | 224x224 | scale, crop, normalize | topK, dequantization* | Y | Y | Y | Y |
| Classification | NasNet Mobile | 331x331 | scale, crop, normalize | topK, dequantization* | Y | N | Y | N |
| Classification | SqueezeNet | 227x227 | scale, crop, normalize | topK, dequantization* | Y | N | Y | N |
| Classification | EfficientNet-Lite0 | 224x224 | scale, crop, normalize | topK, dequantization* | Y | Y | Y | Y |
| Classification | AlexNet | 256×256 | scale, crop, normalize | topK, dequantization* | N | N | Y | Y |
| Face Recognition | Inception v4 | 299 x 299 | scale, crop, normalize | topK, dequantization* | Y | Y | Y | Y |
| Face Recognition | Inception v3 | 299x299 | scale, crop, normalize | topK, dequantization* | Y | Y | Y | Y |
| Segmentation | Deeplab-v3 Mobilenet-v2 | 513x513 | scale, normalize | mask flattening | Y | N | Y | N |
| Object Detection | SSD MobileNet v2 | 300x300 | scale, crop, normalize | topK, dequantization* | Y | Y | Y | Y |
| Pose Estimation | PoseNet | 224x224 | scale, crop, normalize, rotate | calculate keypoints | Y | N | Y | N |
| Language Processing | Mobile BERT | - | tokenization | topK, compute logits | Y | N | Y | N |

TABLE I: Comprehensive list of benchmarks. Each entry shows all possible pre- and post-processing tasks that we observed across our experiments. Tasks marked with an "*" are only performed with quantized models.

with cross-platform frameworks such as NNAPI, each vendor is responsible for implementing the corresponding drivers, an application's performance profile can vary significantly across devices.

### D. Model Execution

This stage encompasses the invocation of a model and retrieval of the results back to the calling application. In frameworks such as TFLite, these steps are handled by a combination of calls to OS APIs such as Android's NNAPI and device specific driver implementations. These drivers can be open-source, such as the TFLite Hexagon Delegate, or bundled into vendor-specific SDKs, such as Samsung's Edge delegate.

In NNAPI, the model execution details are determined during a step called *model compilation*, which usually needs to be performed only once when loading a new model into an application. Based on the application's execution preference (low-power consumption, maximum throughput, fast execution, etc.), the framework will determine (and remember for successive executions) on which processors and co-processors to run a model. Given the abundance of accelerators on modern SoCs, NNAPI tries to distribute a model's execution across several devices. A step referred to as *model partitioning*.

*Offloading* is the process of performing a task (ideally more efficiently) on an attached accelerator instead of the CPU. There are two common models for integrating an accelerator into an SoC. In the tightly coupled model, an accelerator is integrated with the CPU core and its cache hierarchy. In the loosely coupled model, the accelerator is a separate hardware block with its own memory subsystem and communicates with the CPU over memory-mapped I/O [25]. The accelerators in the mobile devices that we consider (namely the Snapdragon SD6xx chipsets) are loosely coupled to the CPU. Thus any communication with the DSP requires a round-trip through the kernel device driver interface.

### E. Post-processing

Post-processing refers to the remaining computations on the model's outputs before presenting them to the user. As with pre-processing algorithms, the details are task-dependent. For image classification tasks that we considered, all that is left to do on the output tensor is pick the most fitting predictions (referred to as *topK*). The outputs of a model are sorted by the likelihood of labels, and so choosing *topK* elements is simply an array slice operation. Other tasks like pose estimation or image segmentation require more intensive data processing on the model output. E.g., an application using PoseNet must map the detected key points to the image.

## III. Experimental Setup

In this section, we summarize our experiments' infrastructure, configuration, and measurement setup and provide the necessary details to reproduce our results.

### A. Machine Learning Models

We measure individual stages of the ML pipeline across a representative set of tasks, input data sizes, and models. Table I summarizes the models we used during our experiments. All models are hosted by TFLite [26]. Implementations for the pre- and post-processing tasks are TensorFlow Lite Android libraries. They are well optimized for production use cases [27].

We picked a set of commonly used models on mobile systems that cover a range of input image resolutions (224x224 to 513x513) and perform pre-/post-processing tasks of varying complexity. We also picked models that are designed for mobile devices (e.g., MobileNet) and more general-purpose models (e.g., Inception). We experiment on two widespread numerical formats for mobile ML: (1) 8-bit quantized integers (INT8) (2) 32-bit floating-point (FP32). Performance and accuracy are often tightly correlated. For our work, accuracy does not matter because we do not compare FP32 against INT8 models. Any comparisons we make are between the same model and bit-width. Hence, we do not discuss model accuracy in the paper.

### B. Software Frameworks

We use the open-source TFLite command-line benchmark utility (and it's Android wrapper) to measure inference pre-processing, post-processing, and inference latency. The former two required additions of timing code around the corresponding parts of the source. By default, the utility generates random

| System | SoC | Accelerators |
|---|---|---|
| Open-Q 835 µSOM | Snapdragon 835 | Adreno 540 GPU, Hexagon 682 DSP |
| Google Pixel 3 | Snapdragon 845 | Adreno 630 GPU, Hexagon 685 DSP |
| Snapdragon 855 HDK | Snapdragon 855 | Adreno 640 GPU, Hexagon 690 DSP |
| Snapdragon 865 HDK | Snapdragon 865 | Adreno 650 GPU, Hexagon 698 DSP |

TABLE II: Platforms we used to conduct our study. We use a `userdebug` build of AOSP in order to get the right instrumentation hooks for our purposes while mimicking the final product build as closely as possible.

tensors as input data. Each model invocation is performed 500 times. Depending on the exact experiment, we schedule the model on the open-source GPU Delegate or Hexagon Delegate, via automatic device assignment by NNAPI, the CPU (across four threads) or a combination of all the above. By default, the benchmarks use NNAPI's default execution preference setting `FAST_SINGLE_ANSWER`.

To evaluate the effect of embedding ML models into mobile applications, we profile a set of open-source applications. For image classification, segmentation, and pose estimation tasks, we utilize TFLite's open-source example Android applications [28]. We have seen similar implementations of the data capture, pre-processing, post-processing, and inference stages across other Android apps.

### C. Hardware Platforms

Table II describes the hardware platforms we study in our characterization evaluation. We focus on the Qualcomm Snapdragon series ranging from SD835 to SD865 as these chipsets are widely deployed in the ecosystem. We only present results on the Google Pixel 3 (SD845), although our experimental results indicate that the trends are representative across the other, older and newer, chipsets. The other chipsets have the same set of hardware accelerators. The chipsets contain a CPU, GPU, and DSP for AI processing tasks. Moreover, these chipsets were used by Qualcomm to generate the MLPerf Inference [19], [29] and recent MLPerf Mobile results [30].

We ran a similar set of experiments through a non-NNAPI framework called Qualcomm's SNPE using open-source applications found on Qualcomm's Developer Network [31] and using the SNPE benchmark tool [32] instead of TFLite's. We came to similar conclusions about end-to-end latency trends, but due to a lack of model variety and available Android apps targeting these systems, we decided to focus our discussion solely on NNAPI. We expect to see similar results across non-Qualcomm SoCs because many of the performance trends we discuss are consequences of *how* frameworks and applications use hardware instead of the underlying SoC hardware itself.

### D. Probe Effect

In our measurements we keep the performance degradation due to our instrumentation infrastructure to a minimum. To decrease variance in our results, we report the arithmetic mean of 500 runs. Since mobile SoCs are particularly susceptible to thermal throttling, we make sure to run benchmarks once the CPU is cooled to its idle temperature of around 33 °C.

When enabling our driver instrumentation, we observe a 4-7% increase in inference time with hardware acceleration enabled (via SNPE or NNAPI) and has no effect on pre-processing or inference performed on the CPU. These increased numbers are not the ones we use in our evaluation since the instrumentation only serves to reveal code paths and measure time spent in drivers. The only results we show that had instrumentation enabled that influenced latency on milliseconds' order is in the offload section (See Section IV-C).

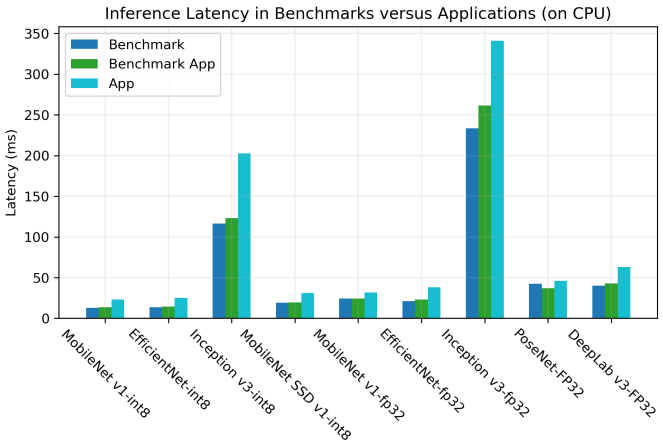
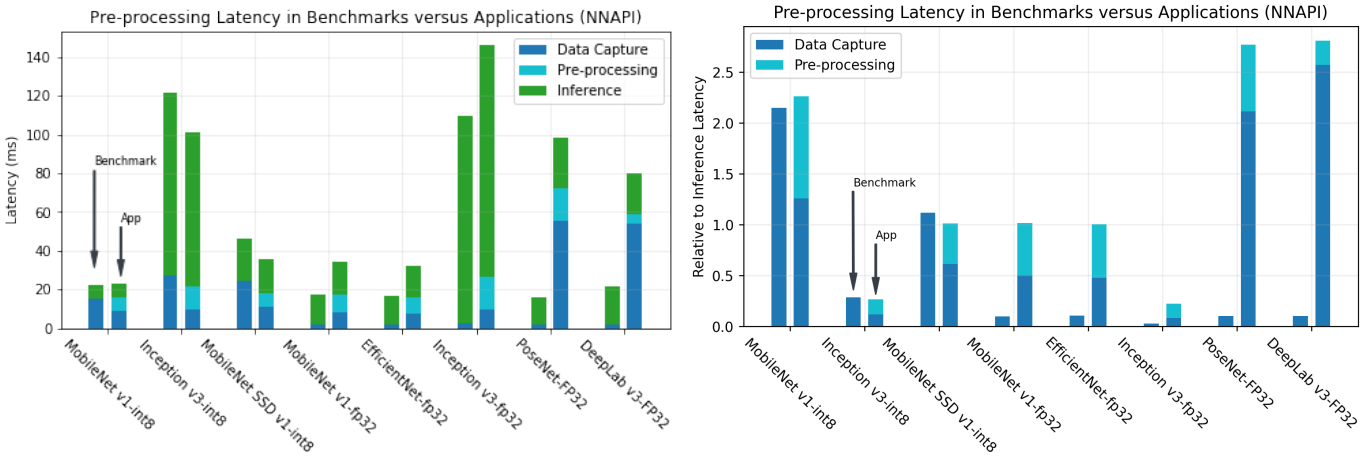



Fig. 3: Comparison of inference latency between the TFLite command-line benchmark utility, TFLite Android benchmark app and example Android applications.

## IV. AI Tax for Mobile

*AI tax* is the time a system spends on tasks that enable the execution of a machine learning model; this is the combined latency of all non-inference ML pipeline stages (defined in section II). We claim that viewing benchmark results and model performance in the context of AI tax can steer the mobile systems community towards fruitful research areas and narrow in on the parts of a system that are sources of performance bottlenecks and need optimization.

To understand and quantify the real-world effects of the end-to-end AI processing, we first compare the performance of ML models when they are run as (1) pure benchmarks from the command line; (2) packaged into benchmark apps with a user interface (TFLite Android benchmark utility [33]); and (3) executed as part of a real application. An example of how benchmarks or proxy applications can deviate from real-world application performance is shown in Figure 3 where we measured end-to-end latency of various models running on the CPU. Since TFLite's benchmark utility focuses on testing inference performance, we observe that negligible pre-processing is required to run a model. Both benchmark utilities (including the one that aims to be more representative of Android applications) masks the end-to-end performance penalties that arise from data capture and pre-processing. For




(a) (b)

Fig. 4: Comparison of the time spent on pre-processing and data capture compared to inference in the *TFLite benchmark utility* as opposed to Android applications. In (a) we report absolute numbers for all three components, while in (b) we report the data capture and pre-procesing latency relative to the inference latency. Compared to benchmarks, the same model encapsulated inside a real application spends a significant amount of time waiting for data capture and pre-processing before it is invoked for inference.

example, in Inception V3-fp32, the app latency is nearly 350 ms while the benchmark latency is 100 ms less at 250 ms. The trend holds true across all the machine learning models.

Consequently, this section aims to convey the importance and demonstrate the identification of AI tax in mobile applications. AI tax manifests itself through three broad categories:

1) Algorithms: how an application pre-/post-processes data and the complexity and input shape of a model.
2) Frameworks: the choice of a machine learning inference runtime to drive the execution of a model.
3) Hardware: availability of hardware accelerators and the mechanisms by which the CPU offloads compute (including any OS-level abstractions).

The following sections present our findings across the categories. We find that the time spent in OS and vendor drivers quickly amortize across hardware delegation frameworks. Together with post-processing, they constitute only a small fraction of the overall execution time. However, the time spent on capturing and then pre-processing data for ML models' consumption can combine to a majority of the execution time.

### A. AI Tax: Algorithms

This section presents our pre-/post-processing and inference latency measurements and shows how the performance patterns differ between benchmarks and Android applications. We measure the time spent on acquiring and pre-processing input data to highlight the differing performance profiles between benchmarks and applications. Although most of our results suggest that post-processing latency is negligible (sub-millisecond per inference), more complex tasks such as image segmentation and object detection show that applications require significant additional work on the model output (e.g., mask flattening and bounding box tracking, respectively).

**Data Capture & Pre-Processing.** Figure 4 contrasts the data capture and pre-processing latency between those measured in inference benchmarks versus Android applications using the same models via NNAPI as the underlying software framework. Figure 4a shows the data capture, pre-processing, and inference latencies, while Figure 4b shows the processing overheads relative to the inference latency to narrow in on details. These show that a significant portion of a model's time is spent on data capture and pre-processing instead of actual inference. That said, the amount of time depends on both the model in question and whether we consider benchmarks or applications.

Models such as quantized MobileNet v1 and SSD MobileNet v1 spent up to two times as much time acquiring and processing data than performing inference in both benchmarks and applications. When embedding floating-point models such as PoseNet, EfficientNet, and DeepLab v3 into applications, we similarly see the majority of time spent outside of inference. On the other hand, in benchmarks, inference latency dominates. Generally, for floating-point data, the "data capture" (which in the case of the inference benchmarks is random data generation) is negligible. In quantized models, it seems to approximate real applications to some extent. However, this is one of the fallacies of this approach to data acquisition. The standard C++ library that this benchmark happened to be compiled against (libc++) generates real numbers significantly faster than integers. Using a different standard library (libstdc++), we observed the exact opposite behavior. The only model where inference latency dominates is for quantized and floating-point Inception-v3. Not only do the Inception models

have significantly more number of parameters and operations than other more mobile-friendly models we test, they are only partially able to be offloaded by NNAPI and runs around half of its inference on the CPU. For image classification and object detection, pre-processing and data acquisition contribute approximately equal amounts to overall run-time, while for PoseNet and Deeplab-v3 pre-processing contributes to about 10% and 1%, respectively.

**Post-Processing.** Many machine learning tasks, particularly image classification, require relatively little post-processing. However, applications in domains such as object detection commonly employ CPU-intensive output transformations *after every inference* to make the output data user-friendly. For example, a typical application using PoseNet maps the detected key points to the image. Also, *Dashcams*, for instance, compute and visualize bounding boxes from a model's output. Similar to pre-processing overheads, a system designer should keep in mind that post-processing can quickly become an application's main bottleneck depending on the task; these patterns again will likely be absent from a benchmark report.

*Takeaway from AI Tax: Algorithms*

When assessing or benchmarking ML system performance, it is crucial to quantify the end-to-end ML system performance that includes the overheads from capturing data along with pre- and post-processing as these can be as much as 50% of the total execution time. Therefore, obsessing about ML-only performance can lead us to miss the forest for the trees.

### B. AI Tax: Software Frameworks

The next set of contributors to ML application overhead are specific to the choice of ML framework. Prominent examples include open-source projects such as TFLite or vendor-specific development kits such as Qualcomm's SNPE or MediaTek's NeuroPilot [14]. These frameworks abstract away most of the execution pipeline of Figure 2, including management of model lifetime, communication with accelerators, and pre- and post-processing primitives' implementation. In this section, we use NNAPI to illustrate the pitfalls in evaluating AI performance if we do not benchmark end-to-end performance.

**Drivers & Scheduling.** Since NNAPI is in large part an interface that relies on mobile vendors to implement, performance bottlenecks can commonly arise due to inefficiencies in the driver implementations, e.g., how models are executed or how specific NNAPI operations are implemented. Figure 5 shows a situation in which this effect is pronounced. We ran the quantized EfficientNet-Lite0 model on four different device targets through TFLite: (1) the open-source Hexagon backend, (2) 4 CPU threads, (3) a single CPU thread, and (4) NNAPI.

Relying on NNAPI's automatic device assignment degrades performance by 7× compared to scheduling the model on a single-threaded CPU. Interestingly this does not occur in the floating-point model, pointing to several subtle traps. This

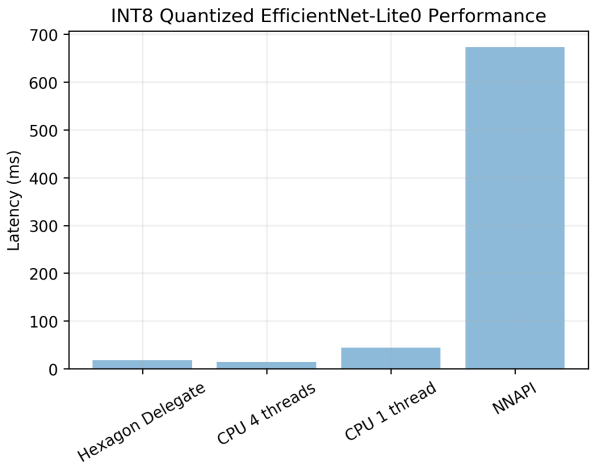



Fig. 5: Performance degradation of TFLite's quantized EfficientNet-Lite0 when using NNAPI (with CPU fallback).

is because of NNAPI driver support, which is lagging for the INT8 operators the model implementation used. Future iterations may likely fix this performance "bug." However, the takeaway is that not all frameworks are created and treated equal.

To root-cause the reason for the overhead, we analyze the execution profile of executing EfficientNet-Lite0 through an image processing app on our Pixel 3. The measured profile is shown in Figure 6. Execution on four CPU threads behaves as expected; cores 4-7 are at 100% utilization for the benchmark (see annotation (1)). Similarly, execution through Hexagon shows 100% utilization of the cDSP and increased AXI traffic (2). However, NNAPI's automatic device selection does not decide on efficient execution. Initially, we see an attempt to communicate via the DSP through a spike in the Compute DSP (CDSP) utilization at the start of the benchmark (5th row from the bottom). Then, the rest of the execution uses a single thread (indicated by the sporadic CPU 100% utilization across cores 4-7 (3)). Moreover, we witness frequent CPU migrations (characterized by more frequent context switches and the core utilization pattern (4)). This failure of offloading to a co-processor and reliance on the OS CPU scheduler ruins inference latency.

While Figure 6 shows a specific case study of a single neural network, similar behavior is observed on other networks. Though the DSP as an accelerator is always supposed to run fast, there is no guarantee that the model will always run faster.

We extended our analysis (not shown) to include the performance of various models running on the CPU (using the native TFLite delegate) versus the DSP (using the NNAPI driver backend). In all cases except for Inception V4, we had observed that the NNAPI-DSP code path is slower. When introspecting the execution using the execution profile, we see similar behavior as in Figure 6—falling back on the CPU results in poor end-to-end system performance.

When we switch the framework to the vendor-optimized Qualcomm SNPE, the DSP's performance is significantly

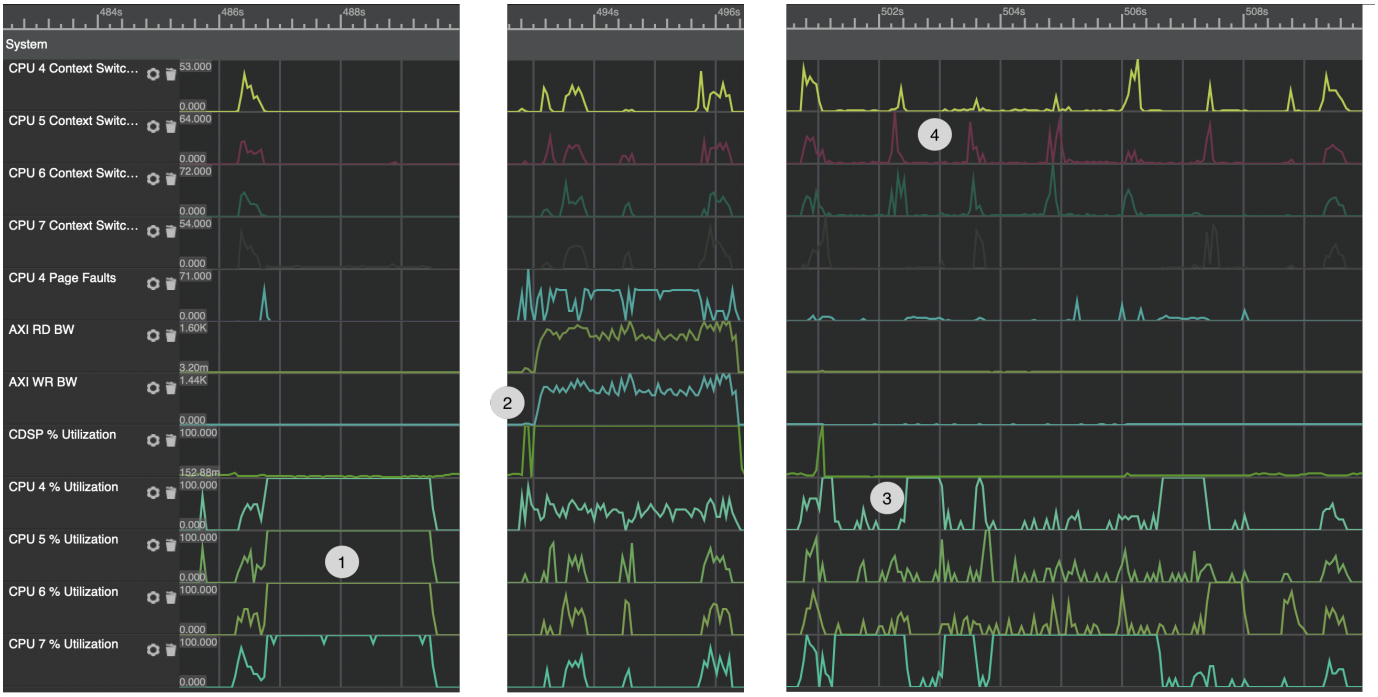



Fig. 6: Output from the *Snapdragon Profiler*, which shows measurements while running the EfficientNet-Lite0 model on the CPU, TFLite Hexagon delegate, and NNAPI. The benchmark was run using TFLite's command-line benchmark utility.

better. The models' performance on the DSP outperforms the CPU (as one would expect). This tells us that being aware of the software framework's choice can greatly impact the performance of the chipset. The SoC vendor-specific software is highly tuned for the SoC and provides optimized support for the neural network operators.

The unfortunate side-effect of vendor-optimized software backends is that it creates silos that can make it difficult for application developers to know which is the right framework for wide-scale deployment. Ahead of time, it is unclear which framework(s) provides the best (and the most flexible) support until the developers (1) download the framework, (2) select a set of ML models to run, and (3) profile those models on the chosen frameworks for their target SoCs.

*Takeaway from AI Tax: Software Frameworks*

Not all frameworks are created equal. Frameworks that poorly support models (at least partially) fallback on the CPU, resulting in worse performance than using the CPU from the start instead of attempting to leverage the promised hardware accelerator performance. Hence, there is a need for greater transparency in frameworks being used during performance analysis.

### C. AI Tax: Hardware Performance

Modern mobile SoCs can include several accelerators for different problem domains, e.g., audio, sensing, image processing, and broadband [34]. The behavior of these accelerators is strongly tied to how the software counterparts invoke them. We assess how invocation frequency affects AI performance.

**Cold start.** We use the Qualcomm Snapdragon 8xx chipsets that include a Hexagon DSP that has been optimized for machine learning applications over the years and is commonly referred to as one of Qualcomm's Neural Processing Units (NPU). It is reminiscent of a VLIW vector processing engine.

Qualcomm developed the Fast Remote Procedure Call (FastRPC) framework to enable fast communication between CPU and DSP. We measure the FastRPC calls themselves, i.e., the overhead of offloading machine learning inference to a DSP. The framework requires two trips through the OS kernel with the FastRPC drivers signaling the other side upon receipt/transmission, as shown in Figure 7. We were interested in how much this contributes to the hardware AI tax. The figure shows the various system-level call boundaries that have to be crossed when offloading to a DSP. Note the cache flush that has to occur to maintain coherency.

Figure 8 shows the measured inference and offloading time for MobileNet v1 when using NNAPI Hexagon delegate for executing the inference task. As we can see, for a small number of inferences, the offloading cost is dominant in the inference latency. However, by increasing the inference operations, the proportion of offloading time to inference time (pre inference) decreases. The DSP initial setup (mapping the DSP to the application process) is done once, and we can perform multiple inferences using the same setup.

Based on the results, the reason for concern is that current

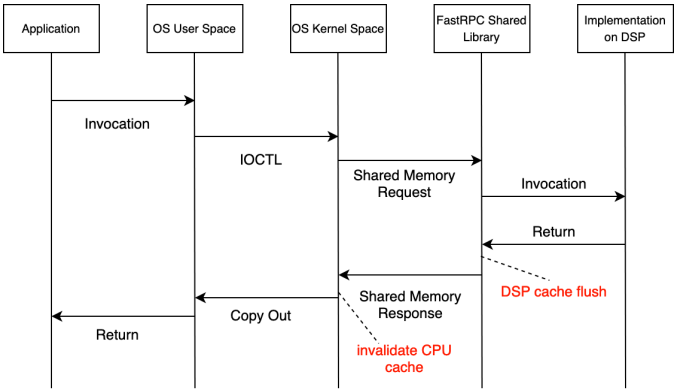



Fig. 7: FastRPC call flow for the Qualcomm DSP.

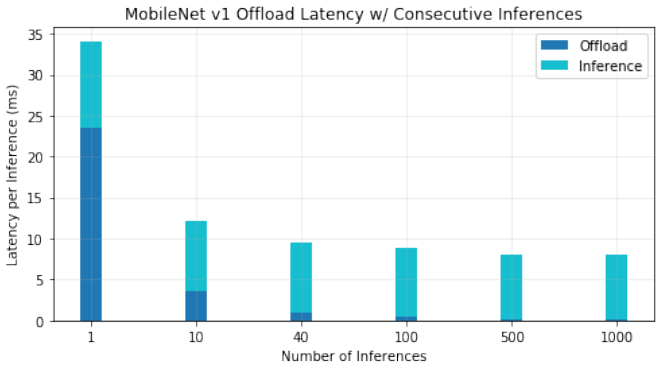



Fig. 8: Overhead amortization over consecutive inferences.

benchmarks and performance analysis often allow for warm-up time that is not necessarily representative of a real-world application. End-user experience, especially in smartphone deployments, involves a cold start penalty that should also be measured in ML benchmarks and workload performance analysis. The TFlite benchmark tool breaks down model initialization time, which is good to measure if an application switches between models or frequently reloads them.

**Multi-tenancy.** An emerging use-case in real-world applications is the growing need to support multiple models running concurrently. Example application use-cases are hand-tracking, depth-tracking, gesture recognition, etc., in AR/VR. Yet, most hardware today supports the execution of one model at a time. To this end, we study AI hardware performance as more tasks are scheduled to the CPU and AI processors.

Figure 9 shows the latency breakdown of our image processing Android application when we schedule increasingly many inferences through the NNAPI Hexagon Delegate in the background (using the TFlite benchmark utility). We observe a linear increase in the latency per inference because the inference is stalled on resource availability. The main inference is running on the DSP but there is only one DSP available for ML model acceleration on this particular SoC. Meanwhile, pre-processing and data capture is approximately constant regardless because activity on the CPU is not affected by the concurrent inferences.

In contrast, Figure 10 breaks down the application latency with the background inferences all on the CPU (while the image processing app still offloads to the DSP). As expected, the time spent on pre-processing and data-capture increases because more tasks are running on the CPU concurrently. Inference latency is now constant because multiple processes are not contending for the DSP.

This simple experiment demonstrates a potential pitfall. Suppose we as system designers or application writers were to focus only on one part of the execution pipeline (e.g., pre-processing in Figure 9 or inference latency in Figure 10) in isolation. In that case, we could wrongly conclude from the results that the device selection and schedule are optimal. However, this need not be the case since we may have other similar applications running concurrently that contend for the same hardware resources and may need to rethink resource allocation, perhaps running one model on the CPU while another uses the DSP or offloading other execution stages in certain circumstances.

While NN accelerators can speed up execution, benchmarks that solely focus on inference latency could mislead SoC designers to reserve silicon area for hardware that speeds up a specific model while eliminating potentially useful other components. For example, DSPs are frequently used for everyday image processing tasks (mainly through computer vision frameworks such as FastCV), while a specially tailored NPU typically cannot support such activities. In such cases, it is potentially more feasible to trade-off a more powerful NPU for a smaller one paired with a DSP for pre-processing, freeing up the CPU to work on other parts of the application.

**Run-to-run Variability.** Performance variability is a major concern for mobile developers because it affects end-user experience. But it is often not accounted for when assessing device performance. This is specifically an area that is overlooked by today's mobile AI benchmarks. Performance on a real device while running applications varies from run to run due to background interference, affecting user experience.

Figure 11 shows performance variation to quantify the gap between benchmarks and mobile AI performance when the same AI models are packaged into a real application form factor. The results show that repeated benchmark executions have a very tight distribution in performance. The deviation from the mean latency is minimal. However, for the models running inside an app, we see a very pronounced distribution in mean latency. The latency can vary by as much as 30% from the median for the benchmarks.

Such large performance variability stems from the non-ML portion of the application code. Data capture, pre-processing, and post-processing all involve multiple hardware units and processing subsystems. To this end, the Android operating system's scheduling decisions, delays in the interrupt handling from sensor input streams, etc., lead to such fluctuations.

Due to such pronounced variability, workload performance analysis needs to report statistical distributions in performance. Instead, today's standard practice is to report a single ML

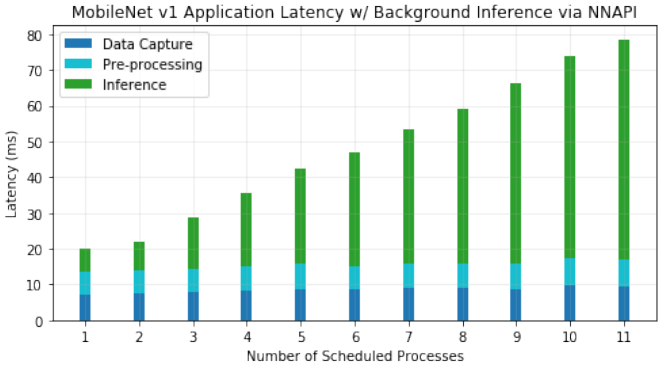



Fig. 9: Latency breakdown of image classification app when scheduling an increasing number of inference benchmarks through NNAPI in the background (using the same model).

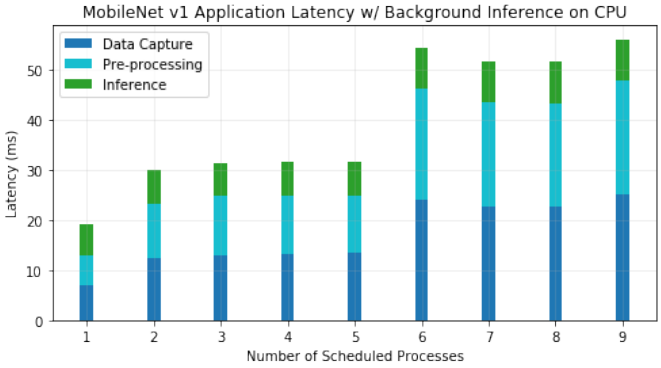



Fig. 10: Same experimental setup as in figure 9 except background inferences are scheduled on the CPU.

performance number, even in well-established standard benchmarks like MLPerf [29], [35], [36]. In prior work, Facebook reported a similar observation in their analysis of mobile inference systems in the wild [37]. There is a greater need for us to model performance variability in workload analysis, which was also recommended in the aforementioned prior art.

*Takeaway from AI Tax: Hardware Performance*

Cold start penalty is common in mobile devices and needs to be considered, yet it is often ignored in ML benchmarks. Moreover, today's benchmarks focus almost exclusively on a single benchmark running in pristine conditions that are not representative of the end-user's operating conditions, which can lead to misguided conclusions about overall system optimization. Finally, run-to-run variability is important to faithfully assess mobile AI performance.

## V. Related Work

**AI/ML Mobile Benchmarks.** There have been many efforts in recent years to benchmark mobile AI performance. These include Xiaomi's Mobile AI Benchmark [38], AI Benchmark [39], AI Mark [40], AITUTU [41], Procyon AI Inference Benchmark [42] and Neural Scope [43]. Nearly all of these benchmarks provide a configurable means to select the software backend, allowing users to use multiple ML hardware delegation frameworks. Many of these benchmarks run a set of pre-selected models of various bit widths (INT8, FP16, or FP32) on the CPU and open-source or vendor-proprietary TFLite delegates. Some of them even enable heterogeneous IP support. Regardless of all these functionalities, they miss the crucial issue of measuring end-to-end performance, which is the ultimate focus of our work. Our objective is to raise awareness of AI tax so that future benchmarks can be extended to be more representative of real-world AI use cases.

**End-to-end AI Performance Analysis.** Kang et al. [44] have investigated the end-to-end performance pipeline of DNNs in server-scale systems and come to a similar conclusion: pre-processing being the main bottleneck in hardware-accelerated models. The authors propose optimizations to common pre-processing steps, including re-ordering, fusion, and datatype changes within the network compilation graph. It features an in-depth analysis of ResNet and proposes a model that is aware of pre-processing overheads to reduce end-to-end latency. We study a more comprehensive range of models and applications on mobile devices. Also, we analyze the results from a systems perspective and provide analyses of ML algorithms, frameworks and hardware.

Similarly, Richins et al. [45] do a study that is specific to datacenter workloads and architectures. It identifies network and storage as the major bottlenecks in the end-to-end datacenter AI-based applications, which commonly include multiple inference stages (e.g., video analytics applications). The authors show that even though their application under consideration is built around state-of-the-art AI and ML algorithms, it relies heavily on pre- and post-processing code, which consumes as much as 40% of the total time of a general-purpose CPU. Our work focuses on mobile SoCs and identifies the non-AI sources of overhead typical for handheld AI devices in the consumer market, which differs from PC and datacenter hardware in the type and number of available accelerators on a given SoC and as of now targets a very different software infrastructure.

## VI. Conclusion

There is a need for end-to-end performance analysis of AI systems. On mobile systems, due to the heterogeneous hardware and software ecosystem, it is essential to have more transparency into the actual end-to-end execution. The AI tax overheads we explicate across algorithms, software frameworks, and hardware highlight new opportunities for existing ML performance assessment efforts to improve their measurement methodologies. Our work motivates characterization studies across more ML frameworks and SoCs with a balanced focus on the entire system and the end-to-end pipeline instead of focusing on ML application kernels only

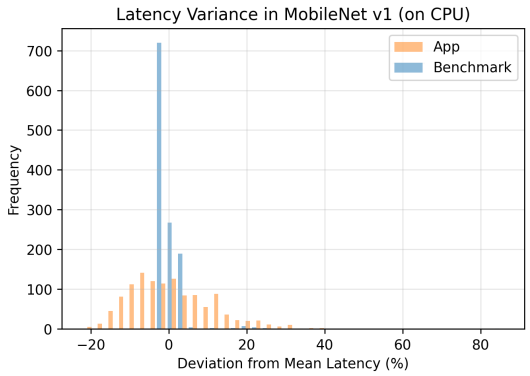



Fig. 11: Latency distribution for image classification usingMobileNet v1 on the CPU. We contrast the distribution inapplications to the TFLite benchmark utility.

in isolation. Such studies will help us develop a more comprehensive performance model that can aid programmers and SoC architects in optimizing bottlenecks in ML applications accelerate the different stages of the ML pipeline.

## ACKNOWLEDGEMENTS

Supported by the Semiconductor Research Corporation.

## REFERENCES

[1] W. Yang, X. Zhang, Y. Tian, W. Wang, J. Xue, and Q. Liao, "Deep learning for single image super-resolution: A brief review," *IEEE Transactions on Multimedia*, vol. 21, no. 12, pp. 3106–3121, 2019.

[2] G. Guo, S. Z. Li, and K. Chan, "Face recognition by support vector machines," in *Proceedings 4th IEEE international conference on automatic face and gesture recognition (cat. no. PR00580)*. IEEE, 2000.

[3] V. Badrinarayanan, A. Kendall, and R. Cipolla, "Segnet: A deep convolutional encoder-decoder architecture for image segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 12, pp. 2481–2495, 2017.

[4] G. Campagna, R. Ramesh, S. Xu, M. Fischer, and M. S. Lam, "Almond: The architecture of an open, crowdsourced, privacy-preserving, programmable virtual assistant," in *Proceedings of the 26th International Conference on World Wide Web*, ser. WWW '17. Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee, 2017, p. 341–350. [Online]. Available: https://doi.org/10.1145/3038912.3052562

[5] F. Seide, G. Li, and D. Yu, "Conversational speech transcription using context-dependent deep neural networks," in *Twelfth annual conference of the international speech communication association*, 2011.

[6] L. Zhang, S. Wang, and B. Liu, "Deep learning for sentiment analysis: A survey," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 8, no. 4, p. e1253, 2018.

[7] P. Molchanov, S. Gupta, K. Kim, and J. Kautz, "Hand gesture recognition with 3d convolutional neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, 2015.

[8] Google coral. [Online]. Available: https://coral.ai/

[9] Mediatek dimensity. [Online]. Available: https://i.mediatek.com/mediatek-5g

[10] Samsung neural processing unit. [Online]. Available: https://news.samsung.com/global/samsung-electronics-introduces-a-high-speed-low-power-npu-solution-for-ai-deep-learning

[11] Qualcomm neural processing engine. [Online]. Available: https://developer.qualcomm.com/docs/snpe/overview.html

[12] Nnapi. [Online]. Available: https://developer.android.com/ndk/guides/neuralnetworks

[13] Snapdragon neural processing engine sdk. [Online]. Available: https://developer.qualcomm.com/docs/snpe/overview.html

[14] Mediatek neuropilot. [Online]. Available: https://www.mediatek.com/innovations/artificial-intelligence

[15] Snapdragon 835 mobile platform. [Online]. Available: https://www.qualcomm.com/products/snapdragon-835-mobile-platform

[16] Snapdragon 845 mobile platform. [Online]. Available: https://www.qualcomm.com/products/snapdragon-845-mobile-platform

[17] Snapdragon 855 mobile platform. [Online]. Available: https://www.qualcomm.com/products/snapdragon-855-mobile-platform

[18] Snapdragon 865+ 5g mobile platform. [Online]. Available: https://www.qualcomm.com/products/snapdragon-865-plus-5g-mobile-platform

[19] P. Mattson, C. Cheng, G. Diamos, C. Coleman, P. Micikevicius, D. Patterson, H. Tang, G.-Y. Wei, P. Bailis, V. Bittorf *et al.*, "Mlperf training benchmark," *Proceedings of Machine Learning and Systems*, vol. 2, pp. 336–349, 2020.

[20] A. Ignatov, R. Timofte, W. Chou, K. Wang, M. Wu, T. Hartley, and L. Van Gool, "Ai benchmark: Running deep neural networks on android smartphones," in *Proceedings of the European Conference on Computer Vision (ECCV) Workshops*, September 2018.

[21] Qualcomm fastcv. [Online]. Available: https://developer.qualcomm.com/software/fastcv-sdk

[22] [Online]. Available: https://developer.android.com/guide/topics/media/camera

[23] [Online]. Available: https://cloud.google.com/tpu/docs/inception-v3-advanced

[24] A. Kendall, M. Grimes, and R. Cipolla, "Posenet: A convolutional network for real-time 6-dof camera relocalization," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2015.

[25] Y. S. Shao and D. Brooks, "Research infrastructures for hardware accelerators," *Synthesis Lectures on Computer Architecture*, vol. 10, no. 4, pp. 1–99, 2015.

[26] [Online]. Available: https://www.tensorflow.org/lite/guide/hosted-models

[27] [Online]. Available: https://github.com/tensorflow/tflite-support/

[28] [Online]. Available: https://github.com/tensorflow/examples/

[29] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou *et al.*, "Mlperf inference benchmark," in *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020.

[30] Mlperf mobile. [Online]. Available: https://github.com/mlperf/mobile_app

[31] [Online]. Available: https://developer.qualcomm.com/

[32] [Online]. Available: https://developer.qualcomm.com/docs/snpe/benchmarking.html

[33] Tensorflow lite benchmark tool. [Online]. Available: https://www.tensorflow.org/lite

[34] Qualcomm Technologies, Inc., *Qualcomm© Hexagon™ DSPUser Guide*, Qualcomm Technologies, Inc.

[35] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou *et al.*, "Mlperf inference benchmark," *arXiv preprint arXiv:1911.02549*, 2019.

[36] V. J. Reddi, D. Kanter, P. Mattson, J. Duke, T. Nguyen, R. Chukka, K. Shiring, K.-S. Tan, M. Charlebois, W. Chou *et al.*, "Mlperf mobile inference benchmark: Why mobile ai benchmarking is hard and what to do about it," *arXiv preprint arXiv:2012.02328*, 2020.

[37] C.-J. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia *et al.*, "Machine learning at facebook: Understanding inference at the edge," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 331–344.

[38] Xiaomi. Mobile ai bench. https://github.com/XiaoMi/mobile-ai-bench.

[39] A. Ignatov, R. Timofte, A. Kulik, S. Yang, K. Wang, F. Baum, M. Wu, L. Xu, and L. V. Gool. (2019) Ai benchmark: All about deep learning on smartphones in 2019.

[40] AIMark. [Online]. Available: https://play.google.com/store/apps/details?id=com.ludashi.aibenchhl=en_US

[41] ANTUTU AI Benchamrk. https://www.antutu.com/en/index.htm.

[42] UL Procyon AI Inference Benchmark. https://benchmarks.ul.com/procyon/ai-inference-benchmark.

[43] NeuralScope Mobile AI Benchmark Suite. https://play.google.com/store/apps/details?id=org.aibench.neuralscope.

[44] D. Kang, A. Mathur, T. Veeramacheneni, P. Bailis, and M. Zaharia, "Jointly optimizing preprocessing and inference for dnn-based visual analytics," *arXiv preprint arXiv:2007.13005*, 2020.

[45] D. Richins, D. Doshi, M. Blackmore, A. T. Nair, N. Pathapati, A. Patel, B. Daguman, D. Dobrijalowski, R. Illikkal, K. Long *et al.*, "Missing the forest for the trees: End-to-end ai application performance in edge data centers," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 515–528.