MicroEdge: A Multi-Tenant Edge Cluster System Architecture for Scalable Camera Processing

Difei Cao*
difei.cao@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

Jinsun Yoo*
jinsun@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

Zhuangdi Xu
xzdandy@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

Enrique Saurez
esaurez@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

Harshit Gupta
harshitg@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

Tushar Krishna
tushar@ece.gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

Umakishore Ramachandran
rama@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

ABSTRACT

With the proliferation of high bandwidth cameras and AR/VR devices, and their increasing use in *situation awareness* applications, *edge* computing is gaining prominence to meet the throughput requirements of such applications. This work focuses on camera applications that perform real-time Machine Learning inferences on camera frames. We find that Machine Learning based camera applications suffer from hardware resource fragmentation due to models under-utilizing or over-utilizing the accelerator. Meanwhile, it is challenging to support fine-grained resource sharing for accelerators such as TPUs because they can only process requests sequentially in a run to completion fashion. We present MicroEdge, a multi-tenant low-cost edge cluster for camera processing applications running at the edge. MicroEdge provides multi-tenancy support for Coral TPUs by extending K3s, an edge-specific distribution of Kubernetes. Through an admission control algorithm, it allows for fractional assignment of TPU resources commensurate with the application pipeline requirements to ensure that the TPUs are fully utilized. Using real-time camera processing applications and a real-world trace, we show that MicroEdge can support up to 2.8× camera streams for a given hardware configuration compared to vanilla K3s, while maintaining scalability and performance requirements.

*Both authors contributed equally to this research.

Middleware '22, November 7–11, 2022, Quebec, QC, Canada

ACM ISBN 978-1-4503-9340-9/22/11…$15.00
https://doi.org/10.1145/3528535.3565254

CCS CONCEPTS

• **Computer systems organization** → **Real-time system architecture**; *Distributed architectures*; • **Computing methodologies** → Computer vision.

KEYWORDS

Edge computing, Camera processing, Machine learning inference, Resource aware scheduling, Edge TPU

ACM Reference Format:
Difei Cao, Jinsun Yoo, Zhuangdi Xu, Enrique Saurez, Harshit Gupta, Tushar Krishna, and Umakishore Ramachandran. 2022. MicroEdge: A Multi-Tenant Edge Cluster System Architecture for Scalable Camera Processing. In *23rd International Middleware Conference (Middleware '22), November 7–11, 2022, Quebec, QC, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3528535.3565254

1 INTRODUCTION

Emerging geo-distributed camera applications such as pedestrian and vehicle density monitoring [3], multi-camera vehicle tracking [17, 38], and vehicle anomaly detection [41] convert camera streams to actionable knowledge. These applications have stringent performance requirements such as low latency and high network bandwidth. Cloud computing has been the workhorse for throughput-oriented applications for the past two decades. However, sending all of the camera streams to the Cloud for processing is suboptimal for several reasons including end-to-end latency, backhaul network stress, lack of actionable content, and privacy and regulatory concerns for the video data. Edge computing [35] extends the Cloud's centralized utility computing model to geographically distributed computational resources which are closer to the source of the data. Meanwhile, the emergence of low-cost computation hardware such as Raspberry Pi [9], NVIDIA Jetson Nano [25], Intel NCS2 [15], and Google Coral TPU [12] allow just-in-time processing of camera streams close to their sources. With such advances in hardware, edge computing is rising as the ideal platform for camera applications. Naturally, it is important to use

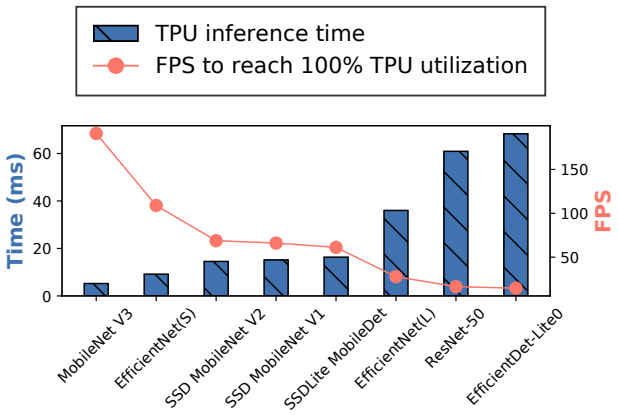


Figure 1: Model Processing Times on TPU. An illustration of the diversity for processing times on the TPU for four detection and four classification models [37]. The orange line depicts the workload (*i.e.*, frame rate) needed for 100% utilization of the TPU for a given model.

these resources efficiently when deploying camera applications at scale.

Motivation and Prior Art. In this work, we develop and optimize a low-cost edge cluster for real-time camera processing called MicroEdge. MicroEdge is composed of Raspberry Pi's (*abbrev.*, RPis) and Coral TPUs (*abbrev.*, TPUs). When deploying camera processing applications at scale, one would have to assign physical hardware resources to each of the camera streams. The seemingly straightforward option is to dedicate hardware resources for every camera stream instance. For example, an exemplar real-time multi-camera vehicle tracking system called Coral-Pie [38] dedicates two RPis and one TPU for every camera stream. In their design, the first RPi and the TPU are responsible for detecting vehicles in the field of view of the camera, while the second RPi re-identifies the detected vehicles based on information from up-stream cameras and notifies downstream cameras for trajectory construction.. They show that with such a careful pipelining of the RPis and TPUs they can meet the application's *service level objective(SLO)* of 15FPS for *SSD MobilenetV2*[34] dataset. However, dedicating hardware resource for a camera stream would lead to under-utilization. For example, for the same setup used in Coral-Pie [38], our study shows that processing a single 15FPS camera stream leads to a mere 30% TPU utilization, thus wasting precious TPU resources. Furthermore, when we introduced the difference detector of NoScope [19] into Coral-Pie's object detection pipeline the TPU utilization drops to 20%. The main takeaway is that the edge cluster would be able to serve several more camera streams if a TPU can be shared across multiple camera streams.

To illustrate the generality of the fragmentation problem, we profile the inference times for 8 pre-trained models [37] and show the results in Fig. 1. The orange line in Fig. 1 shows that to fully utilize a TPU (*i.e.*, 100%), five of the eight models would have to be fed a workload of over > 50FPS. Such a high frame rate is neither warranted nor practical. The average FPS required for surveillance applications using camera networks is around 15 FPS, which is relatively low[16]. The conventional wisdom is that a 15 FPS frame rate delivers the required false positive and false negative metrics for such applications while keeping the cost for the camera networks and resource requirements (network bandwidth, computation, and storage) low. On the other hand, the inference time for a few expensive models (*e.g.*, ResNet-50 and EfficientDet-Lite0 in Fig. 1) may exceed the inter-arrival time between camera frames even at 15 FPS. To process such models, multiple TPUs would need to be assigned to handle a single camera stream. For example, per-frame inference processing for the *EfficientNet-Lite0* model on a TPU takes 69ms. On the other hand, to sustain a frame rate of 15FPS, each frame processing should take less than 66ms. Therefore, to sustain the desired frame rate, the camera stream would have to be partitioned between 2 TPUs, with each TPU handling every other frame. With such a workload partitioning, the utilization on each TPU would be only 52%, thus wasting a cumulative 96% of the available TPU processing power. In other words, even for heavy-weight models requiring higher TPU usage, there would be TPU resource fragmentation when an integral number of TPUs are dedicated to each camera stream. Thus, there is an opportunity to increase resource utilization by eliminating internal fragmentation through fine-grained scheduling and sharing of a TPU across distinct camera streams. The problem of resource fragmentation is general and applies to other accelerator platforms with no native virtualization support.

To address the issue of resource *internal* fragmentation, the key challenge is to facilitate *granular* sharing of TPUs across multiple camera streams while adhering to the performance SLOs for each stream. GPUs enjoy virtualization and sharing support from vendors, such as NVIDIA Docker [27], NVIDIA MPS [24] and NVIDIA Triton [28]; however, other DNN accelerators such as TPUs do not yet have native support for serving multiple applications concurrently. State-of-the-art orchestration systems such as K3s [2], which is a lightweight version of Kubernetes [22] optimized for IoT & Edge computing, do not facilitate fractional sharing of TPUs. There is prior art for scheduling inference requests from multiple applications on GPUs in the Cloud while meeting applicaiton SLOs [7, 14, 33, 40]. Such prior work chooses a *serverless* design approach wherein all requests are forwarded to a per-model shared queue, and scheduling decisions (*e.g.*, deadline driven) are made at runtime based on the priority of the inference invocation. However, such a solution would not be viable in a low-cost edge cluster comprising of RPis and TPUs, wherein the additional data movement due to shared queues and runtime scheduling decisions are detrimental to meeting application SLOs.

Our Approach. To facilitate fine-grained accelerator sharing, we choose Coral TPU, one of the most representative DNN accelerators designed for edge computing, as an exemplar. MicroEdge elevates TPU as a *first class citizen* by extending K3s to orchestrate the resource allocation for the multi-tenant applications (*i.e.*, pods) executing on an edge cluster. In contrast to serverless designs ([14, 28, 39]) which allocate resources at runtime for each function call, MicroEdge allocates TPU resources to collocated camera applications at *deployment* time to avoid runtime overheads. Specifically, MicroEdge ❶ exposes every physical TPU as a *service*, which serializes concurrent TPU requests from multiple application pods. ❷ proposes a new metric for resource specification called "TPU

Figure 2: Camera Applications – A generic pipeline model.

Units", which quantifies the fractional percentage of TPU resource that an application pod needs, and ❸ extends the admission control algorithm in K3s to orchestrate TPU resources upon the creation and destruction of application pods.

Contributions. We make the following contributions described in the rest of the paper:

- We present MicroEdge, a low-cost edge cluster composed of RPis and TPUs that serves the computational needs of geo-local camera applications. § 3 presents the performant multi-tenancy architecture of MicroEdge that facilitates sharing CPU, TPU, and memory resources among various camera applications.
- We propose a new resource specification metric dubbed *TPU units*, which helps to identify TPU resource fragmentation caused by dedicating TPUs to individual applications. Using TPU units as a scheduling parameter, we extend the control plane of K3s to allocate the required TPU resources to application pods as necessary. § 4 extends admission control algorithm which features two techniques, namely, *model co-compiling* and *workload partitioning*, to further reduce the TPU resource fragmentation.
- To support the fine-grained TPU sharing policies of the control plane, we extend K3s' data plane via three components: the *TPU Service*, the *load balancing service*, and the *TPU Client*. These components are elaborated in § 5.
- We evaluate MicroEdge with real-world camera applications which perform multi-camera vehicle tracking [38] and real-time person segmentation [30], and synthetic workloads generated from Microsoft function traces [36]. The performance results in § 6 show that compared to a non-virtualized bare-metal deployment, MicroEdge can more than double the TPU resource utilization and reduce the cost of the Edge cluster by 33%, while meeting camera applications' FPS and latency requirements.

2 BACKGROUND AND ASSUMPTIONS

Camera Applications. Fig. 2 depicts a generic pipeline model for a camera application. The application generates a stream of camera images at a rate commensurate with the application's SLOs. The first stage of the pipeline pre-processes the frames; an example of such pre-processing would be to resize the image frames to fit the machine learning (ML) model used by the subsequent ML inferencing stage. The next stage performs the actual inference, such as detection, classification, and segmentation on the pre-processed camera frame. The final stage performs post-processing depending on the application logic; *e.g.*, the re-identification step in a space-time vehicle tracking application mentioned in § 1.

Meeting the processing throughput requirement in FPS is an important SLO for supporting camera applications; otherwise, the queue build-up of the yet-to-be processed frames will ultimately violate the per-frame processing latency bound for the application.

While cameras produce frames 24 x 7, camera applications may not need to process every frame. For example, to track suspicious vehicles on a geo-deployed camera network, a downstream camera needs to request resources and start processing the camera frames only upon notification of a suspicious vehicle by an upstream camera. The camera will stop processing frames as soon the suspicious vehicle leaves its field of view. In this way, a camera can *utilize* a TPU and process frames only for a short period of time. Therefore, a resource allocator for camera applications at the edge should be capable of allocating and reclaiming resources on a need basis rather than dedicating them for the entire lifetime of each camera stream. While camera streams may dynamically 'come and go' acquiring and releasing resources, specs such as image resolution and FPS are constant throughout the lifetime of a camera stream. Therefore, from the edge cluster's point of view we assume workloads where the number of application instances (*i.e.*, camera streams) may change over time, but the input rate is either provided by the developer or gleaned by profiling before the start of the application.

ML Inference Systems. Prior work for orchestrating GPU resources for ML in the cloud [7, 14, 33] choose a serverless architecture, where scheduling decisions are made per inference request at runtime. MicroEdge on the other hand does *admission control* at deployment time for two reasons: First, since the input rate is known before an application is launched, MicroEdge can reject the deployment request of an application if there are insufficient TPU resources to meet the application SLO; a side benefit of such admission control compared to a serverless design is avoiding wastage of the CPU resources to which an accelerator is attached. Second, MicroEdge avoids the extra data movement per frame and scheduling operations in a serverless design which leads to non-negligible latency overhead for low-cost computational devices (§ 6.4.2).

Container Orchestration System. Kubernetes [22] is a popular container orchestration system for automating the deployment, scaling, and the management of containerized applications. K3s, a purpose-built distribution of Kubernetes for the IoT environment, has fewer extensions and more light-weighted components that are ideal for a resource-constrained edge cluster. K3s is also optimized for the ARM architecture, which forms the core of the RPi processor. K3s deploys an application in a *pod*, which is the smallest unit of deployment. Similar to Kubernetes, K3s supports *labeling* that allows application pods to request nodes with specific features (*e.g.*, a node that has a TPU attached). K3s also supports anti-affinity [20] which would prevent multiple application pods from requesting the same physical node. Put succinctly, K3s is an appropriate starting point for MicroEdge compared to the full-feature Kubernetes. However, the fact remains that neither the function-rich Kubernetes nor K3s provide facilities for granular TPU sharing across camera streams or workload partitioning for a given camera stream.

AI Accelerators in Edge Computing. With growing interest in IoT and Edge computing, various low-cost accelerators for the Edge have emerged such as Google's Coral TPU [12], Intel's NCS2 [15], and NVIDIA's Jetson Nano [25]. In MicroEdge, we equip RPis with Coral TPUs. However, the designs proposed in this work can also be applied to other accelerators. Coral TPU comes with a co-compiling feature [10], which allows transitioning between multiple ML models pre-loaded into the TPU memory. Typically, switching between models on a TPU requires swapping

the new model into the TPU memory resulting in higher latency; on the other hand, co-compiling allows for loading multiple models at the same time into the TPU memory. If the cumulative memory requirement of the co-compiled models exceeds the available TPU memory, models with low priority are partially loaded into the TPU memory, and the remaining portion is loaded at runtime from the host memory. Nevertheless, co-compiling is still faster compared to swapping models in and out of the TPU memory. In § 4, we elaborate on utilizing this co-compiling feature to improve the gain in utilization from TPU sharing.

3 MICROEDGE ARCHITECTURE

The hardware base for MicroEdge is a fairly generic compute cluster comprising RPi 4 CPUs interconnected by two high-speed 16-port Ethernet switches. Each RPi is equipped with a 1 Gigabit NIC to connect to the switch. A portion of RPis are augmented with Coral TPU accelerators through their respective USB ports. The current configuration of MicroEdge supports 25 RPis, out of which 6 of them are endowed with attached TPUs. These hardware resources are grouped into two categories: *vanilla RPis* denoted *vRPis*, and *RPis endowed with TPUs* denoted *tRPis*. The current configuration could be grouped into 19 *vRPis* and 6 *tRPis*. The novel contributions of our work are the control plane and data plane techniques that enable sharing the limited TPU resources of MicroEdge across independent application pipelines and partitioning the requests from a given pipeline onto multiple TPUs to meet the application SLOs. While these techniques are general and can be married into any orchestrator framework, we have chosen to incorporate them into K3s for reasons mentioned in § 2. To set the context for describing the control plane and data plane techniques in § 4 and § 5, respectively, we first present the overall system architecture of MicroEdge shown in Fig. 3.

The K3s scheduler is part of the system software that sits on top of a remote server. The right half of Fig. 3 shows how clients interact with the K3s scheduler to deploy application pods. The left half of Fig. 3 shows the data plane actions once an application pod has been deployed and is executing the application pipeline (similar to Fig. 2). One of the novel contributions of MicroEdge is the extension of K3s's control plane (the box labeled "Ext Scheduler" in Fig. 3) that elevates TPU as a *first class citizen* from the point of view of scheduling. When deploying application pods on *RPis* (both *vRPis* and *tRPis*), the default scheduler of K3s consider *only* CPU and memory resources commensurate with the client requests. It is the extended scheduler that considers TPU resources. As part of system initialization, an entity dubbed *TPU Service* (to be described in § 5.1) is created on each tRPi. The extended scheduler interacts with the TPU Service to make TPU allocations and for loading ML models into the TPU memory. In § 3.1, we describe the workflow in processing a client deployment request to allocate TPU resources in creating the application pod. The second novel contribution of MicroEdge is the set of components added to the data plane of the system stack (left half of Fig. 3) that facilitates the control plane decisions in terms of fractional allocation of TPU resources and workload partitioning. In § 3.2, we describe the data plane workflow when the application pod is executing.

3.1 Workflow of the Control Plane

In this subsection, we present the workflow of the control plane of MicroEdge shown in the right half of Fig. 3. The steps below correspond to the numbered arrows in Fig. 3.

❶: The first step is identical to how a client will interact with K3s for requesting an application pod deployment. The application requirements (such as container image and resource specifications) are presented in a Yaml file. The resource requirements include CPU, memory, and TPU needed for the application pod. K3s performs the default actions to handle the CPU and memory resource requirements and chooses a list of candidate nodes from the pool of *RPis* to host the application pod. It then passes the request to the extended scheduler component (§ 4.1) for TPU allocation contained in the client's resource specification.

❷: The extended scheduler allocates the TPU resources from the pool of *tRPis* based on the client request (*i.e.*, requested models and the amount of TPU resources in the Yaml file). It then loads the requested models on the chosen TPUs. If other models (requested by previous application pods) have already been loaded on a chosen TPU, the extended scheduler invokes the Co-compiler to create a co-compiled model. We elaborate on the admission control algorithm in § 4.

❸: The extended scheduler returns the TPU scheduling decisions to K3s, which then spawns the application pods on the allocated hardware resources of MicroEdge.

❹: When initializing the application pod, the extended scheduler configures the load balancing service (box labeled "LB Service" in Fig. 3) that is attached to the respective TPU Client. LB Service (to be described in § 5.3) is a component in MicroEdge's data plane which is baked into the application pod so that the inference requests from that application pod are routed to the appropriate TPU Service instances (as shown in Fig. 3).

❺: The reclamation component periodically polls the status of application pods. When a pod is terminated, the reclamation component reclaims the associated TPU(s) while the native K3s takes care of reclaiming the CPU and memory resources.

The above control plane workflow is a one-time admission control action. Once an application pod has been deployed on the hardware resources, the data plane(left half of Fig. 3) carries out the necessary application action without involving the control plane.

3.2 Workflow of the Data Plane

The left half of Fig. 3 illustrates the data plane actions when the application pod is executing. Three entities come together to deliver on the admission control decisions taken by the extended scheduler: *TPU Client*, *LB Service*, and, *TPU Service*. TPU Client is a Python library that developers include in creating their applications. It offers the *Invoke* primitive used by the applications to execute inference requests on the TPUs allocated to this application pod. This library relieves the developer from the details specific to a particular invocation such as resizing the raw image to match the required input size of the ML models or establishing connections with the correct TPU Service instance and sending the resized image frames to that instance. As mentioned before, TPU Service is instantiated at system initialization time on every tRPi. It implements two primitives: *Invoke* and *Load*. The former is used by the TPU Client to launch

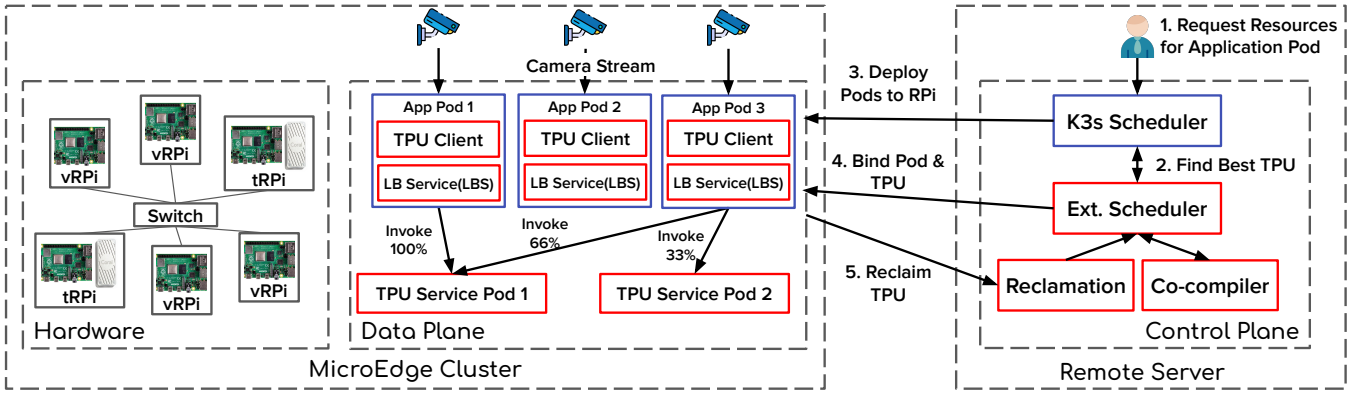


Figure 3: System Architecture for the MicroEdge cluster – MicroEdge is built on Raspberry Pi 4's and Coral TPUs. MicroEdge uses K3s to orchestrate resources for containers (pods). It further extends the K3s' control plane and data plane to manage TPU resources for every pod in addition to CPU and memory resources. The extensions are colored in red in the figure.

an inference request on the associated TPU, while the latter is used by the extended scheduler to load ML models into the associated TPU memory. As mentioned before, LB Service is a component that is configured into the application pod by the extended scheduler as part of admission control. As shown in Fig. 3, this component is responsible for *fanning out* the *Invoke* requests to the appropriate TPU Service instances as per the weights assigned by the extended scheduler. We will elaborate on the design decisions of these three components in § 5.

4 MICROEDGE CONTROL PLANE

As we mentioned earlier, MicroEdge elevates the TPU from a mere peripheral device to a first-class citizen from a resource scheduling perspective. State-of-the-art orchestrators such as K3s fall short of this aspirational goal of MicroEdge, which is the opportunity that we have capitalized on to add to the state-of-the-art through the extended scheduler of MicroEdge. To live within the already existing ecosystem for resource orchestration, we have chosen to *extend* the capabilities of K3s. Thus, we leave the scheduling of CPU and memory to the default capabilities already present in K3s, and invent new techniques for scheduling TPU resources and incorporating them into the K3s ecosystem. In this section, we elaborate on the design decisions of MicroEdge's control plane. These decisions are baked into the extended scheduler. Specifically, § 4.1 describes extensions to the resource specifications submitted to K3s that allow clients to request TPU resources in addition to CPU and memory. § 4.2 presents the algorithm for granular allocation of TPU resources to satisfy client requests. Finally, § 4.3 presents enhancements to the base allocation algorithm for partitioning the workload and fanning out inference requests from a given client to multiple TPU Service instances to meet application SLOs.

4.1 MicroEdge's Pod Creation Interface

To create application pods in K3s, a client needs to specify configuration knobs (*e.g.*, container image, CPU, and memory requirements) via a Yaml file. MicroEdge expands the configuration by two extra knobs to orchestrate TPU resources for the application pods:

- **Model.** The client can specify the inference model to be used in the application. This prior knowledge is critical for MicroEdge's extended scheduler to maximize the resource utilization of the cluster and ensure the performance of application pods. For example, when two application pods request the same model, they can share the same TPU Service instance without paying model switching overhead (*i.e.*, time to load the model into the TPU memory from the host). Furthermore, the extended scheduler infers the size of a model's parameter data. When the total size of multiple models is less than 6.9 MB[1], the extended scheduler can co-compile those models and load the co-compiled models on one instance of the TPU Service (*i.e.*, Space Sharing of TPU Service). Though the TPU executes one inference request fully before entertaining the next one, this co-compilation optimization reduces the model switching overhead.
- **TPU Units.** Similar to the terminology "CPU units" [21] in K3s, we define TPU units. It allows the client to specify the fractional amount of TPU resource that an application pod requires. TPU unit is the *duty cycle* of inference requests that an application pod is expected to generate. More formally, if an application requires an inference service that takes t time units to complete (including model switching time), and the inter-arrival period for successive requests from that application is T time units, then the TPU Unit needed for this application pod is: $t \div T$.
  For example, the TPU Unit required for a camera stream operating at 10 FPS (*i.e.*, 100ms interval between frames) with a per-frame inference service time of 30ms is $0.3 (30 \div 100)$. In other words, the same instance of a TPU Service can be shared across multiple application pods so long as the cumulative TPU Units for all of them is ≤ 1. MicroEdge offers an offline service for a client to profile the inference service time to determine the TPU unit to specify in their request Yaml file.

[1]The TPU has roughly 8 MB of memory that can house the model's parameter data. However, a small amount of that memory is reserved for the model's inference executable, so the parameter data can be housed in the remaining space [11].

4.2 Admission Control Algorithm

Problem Definition. Given an application pod creation request, the extended scheduler needs to assign the TPU resources (through TPU Service) to the pod following two rules:

- TPU Units Rule: the cumulative TPU units assigned to a single TPU should be smaller than 1. This rule guarantees that no TPU is oversubscribed, and all TPUs can finish the inference tasks assigned to them on time.
- Model Size Rule: the cumulative size of distinct models' parameter data loaded on each TPU should be smaller than the total available memory in a TPU. This rule avoids the model swapping overhead through co-compiling when consecutive Invoke calls routed to a given TPU request different models.

The application pod creation request will be rejected[2] when the extended scheduler cannot assign TPU resources to the pod. The objective is to maximize the number of application pod creation requests accepted in MicroEdge.

Bin Packing Problem. We formalize the admission control problem as a bin packing problem [6] with additional constraints on the cumulative size of distinct models loaded on each TPU. In the bin packing problem, items of different sizes must be packed into a finite number of bins, each of a fixed capacity, in a way that minimizes the number of bins used. In MicroEdge, we can view bins as TPUs, each of which has a capacity of 1 TPU unit, items as requested models, and the size of each item as the TPU units requested by the application. For admission control in MicroEdge, the extended scheduler does not have the privilege to reject an application pod for the purpose of reserving the resources for future application pods. In other words, the extended scheduler should try its best to allocate resources for every application pod creation request. Therefore, minimizing the number of TPUs used is equal to maximizing the number of pod creation requests accepted. If the minimum number of TPUs needed is larger than the number of available TPUs, the pod creation request should be rejected. Fig. 4 gives a mathematical formulation of the admission control problem.

Online Heuristics. MicroEdge does not know what application pods may arrive in the future, so we consider an online version of the bin packing problem, where the items arrive one after another and the (irreversible) decision of where to place an item has to be made before knowing the next item or even if there will be another one. Several heuristics-based algorithms are available for the online bin packing problem [6] — Next-Fit, Next-k-Fit, First-Fit, Best-Fit, and Worst-Fit. We extend the First-Fit algorithm for TPU resource scheduling in MicroEdge, which provides an asymptotic approximation ratio of 1.7[3].

$$\min \quad K = \sum_{j=1}^{M} y_j \tag{1}$$

$$\text{s.t.} \quad \sum_{i=1}^{N} t_i x_{ij} \le y_j, \qquad \forall j \in \{1, \dots, M\} \tag{2}$$

$$\sum_{g=1}^{G} s_g \Big(\prod_{i=1}^{N} (\mathbb{1}_{ig} x_{ij})^{c}\Big)^{c} \le 6.9, \qquad \forall j \in \{1, \dots, M\} \tag{3}$$

$$\sum_{j=1}^{M} x_{ij} = 1, \qquad \forall i \in \{1, \dots, N\} \tag{4}$$

$$y_j \in \{0, 1\}, x_{ij} \in \{0, 1\}$$

Figure 4: Mathematical Formulation of Admission Control – M is the number of TPU services, $y_j = 1$ if the j-th TPU is used (*i.e.*, there exists any application pod sending inference requests to the j-th TPU). Eq. (2) defines the TPU Units Rule. N is the number of application pods, t_i is the TPU unit requested by the i-th pod, and $x_{ij} = 1$ if the i-th application pod is allocated with the j-th TPU for inference. Eq. (3) defines the Model Size Rule, where s_g is the size of model $g (g \in [1, G])$. $\mathbb{1}_{ig} = 1$ if the i-th application requests g-th model, $\mathbb{1}^{c}_{ig}$ is the complement of $\mathbb{1}_{ig} = 1 - \mathbb{1}_{ig}$. Eq. (4) makes this a static problem by allocating only one TPU to each application pod. Workload Partitioning (§ 4.3) relaxes the problem by allowing x_{ij} to be a fraction between $\{0, 1\}$.

Algorithm 1 presents the First-Fit-based TPU resources scheduling in MicroEdge. Line 3 shows the TPU unit check specified in Eq. (2). Line 4 shows the model's parameter data size check, which requires that either the model has been already loaded on the chosen TPU or the model's parameter data size is smaller than the available memory of the chosen TPU (Eq. (3)). Line 6 represents the co-compiling procedure when the requested model is not already present on the chosen TPU.

The extended scheduler uses the output of the `AdmissionControl` (*i.e.*, the allocated TPU set and the TPU units are assigned to each TPU in the set) to configure the LBS for the application pod.

Since resource placement is a one time action, the main consideration for scalability is the execution time for Algorithm 1. The complexity of this algorithm is $O(M)$ where M is the number of TPUs. Given space and energy considerations, we do not expect an edge cluster to have more than 100 nodes. Within such realistic edge cluster assumptions, application pod deployment is scalable as shown in the evaluation section (§ 6.2).

Resource Reclamation. An application pod will eventually complete its execution on the TPUs assigned to it. When the Reclamation component in Fig. 3 detects that an application pod is no longer alive, it subtracts the pod's requested TPU units from the `CurrentLoad` of TPUs assigned to the pod. The model reclamation happens lazily in MicroEdge. The extended scheduler subtracts the reference count of the requested model on TPUs assigned to the pod. Later, when the extended scheduler co-compiles the models on TPUs, it excludes those models with a reference count equal to zero.

[2]By default, K3s will also reject application pod creation requests if MicroEdge does not have sufficient CPU and memory resources. For the sake of the discussion on the admission control of TPU resources, we assume CPU and memory resources are sufficient.

[3]The best asymptotic approximation ratio for single-class algorithms is 1.7. Better approximation ratios are possible with refined algorithms [6]. Their explorations are beyond the scope of the paper.

Algorithm 1: Admission Control Algorithm

```
Input  : ApplicationPod: the application pod making the resource request
         Model: the model that the application pod is requesting
         TPUUnit: the amount of TPU time cycles the application pod is requesting
Output : AllocatedTPUs: the physical TPU(s) (through TPU Service) allocated to the
         application pod
         AllocatedTPUUnits: the time cycles on each TPU allocated to the application pod
1  Procedure AdmissionControl (ApplicationPod, Model, TPUUnit)
2      foreach TPU in MicroEdge do
3          if CurrentLoad (TPU) + TPUUnit ≤ 1 then
4              if Model in TPU or ModelSize (Model) ≤ FreeMem (TPU) then
5                  if Model not in TPU then
6                      CoCompile (TPU, Model)
7                  return (TPU, TPUUnit)
8      return None
9  Procedure AdmissionControlWithWorkloadPartitioning (ApplicationPod, Model,
       TPUUnit)
10     if AdmissionControl (ApplicationPod, Model, TPUUnit) is None then
11         AllocatedTPUs ← []
12         AllocatedTPUUnits ← []
13         foreach TPU in MicroEdge do
14             if Model in TPU or ModelSize (Model) ≤ FreeMem (TPU) then
15                 WP ← Min (TPUUnit, 1 - CurrentLoad (TPU))
16                 if WP > 0 then
17                     Append (AllocatedTPUs, TPU)
18                     Append (AllocatedTPUUnits, WP)
19                     TPUUnit ← TPUUnit- WP
20                 if TPUUnit == 0 then
21                     break
22         if TPUUnit > 0 then
23             return None
24         else
25             foreach TPU in AllocatedTPUs do
26                 if Model not in TPU then
27                     CoCompile (TPU, Model)
28             return (AllocatedTPUs, AllocatedTPUUnits)
```

4.3 Admission Control with Fine-Grained Workload Partitioning

Motivation. In the traditional online bin packing problem, we need to decide on which bin (*i.e.*, TPU) to place every application pod upon arrival, which implies that every application pod will be placed into exactly one bin (*i.e.*, x_{ij} in Eq. (2) is an integer of 0 or 1). However, this property leads to TPU resource fragmentation in MicroEdge. For example, consider three application pods requesting the same model and 0.6 TPU units. Under `AdmissionControl` in Algorithm 1, any of two application pods cannot share a single TPU because, by definition of TPU units, the total 1.2 TPU units (by two application pods) exceed the capability of a single TPU. As a result, the extended scheduler allocates a dedicated TPU for each of the three application pods.

Fine-Grained Workload Partitioning. We introduce the workload partitioning mechanism in TPU resource scheduling to address this limitation. As mentioned in § 3, the LBS associated with a specific application pod is designed to send the inference requests from that pod to one of the multiple instances of TPU Service given a list of weights. For example, with reference to Fig. 3, assume all three application pods require 0.6 TPU units. The extended scheduler would allocate TPU Service 1 to application 1; thus, application 1 will send all of its requests to TPU Service 1. There are still 0.4 TPU units available in TPU Service 1. The extended scheduler could allocate the remaining 0.4 TPU units of TPU Service 1 to application 2. The balance of 0.2 TPU units needed for application 2 could be allocated to TPU Service 2. As part of admission control, the extended scheduler will initialize the LBS associated with application pod 2 with the appropriate weights such that at runtime that LBS will send 66% (0.4/0.6) of the inference requests to TPU Service 1 and the balance to TPU Service 2. Finally, TPU Service 2 would also be assigned to application 3 to satisfy its need for 0.6 TPU units. Thus, the TPU resource requests of all three application pods can be met with two instances of the TPU Service.

The procedure of splitting a requested model's TPU unit into multiple smaller portions is dubbed *workload partitioning* in MicroEdge. As part of admission control, the extended scheduler determines the partitioning weights and initializes the LBS associated with the newly created application pod. It is important to note that a *given inference request* is fully executed on the same TPU. Workload partitioning is a way by which *successive requests* emanating from a given application pod can be fanned out to multiple TPU Service instances. This workload partitioning is a powerful technique for ensuring there is no internal fragmentation of TPU resources. Further, this mechanism would also help applications that require more than 1 TPU unit (*e.g.*, ResNet-50 and EfficientDet-Lite0 in Fig. 1).

Lines 9 to 28 presents the workload partitioning mechanism in MicroEdge. The algorithm determines a candidate set of TPUs (if all the requested TPU units cannot be assigned to a single TPU) to serve the inference requests of a new pod that is being created. The algorithm first checks if a candidate TPU either already has the model for the newly created application pod in its memory, or has space in its memory to accommodate the model(Line 14). Such a TPU is included in the allocated TPU set if it also has some spare TPU units to host new requests. Lines 15 to 19 shows the algorithm fragment that determines the workload partition that can be assigned to each TPU in the allocated TPU set. If the model is not already in the memory of an allocated TPU then the algorithm calls the co-compilation procedure (Line 27). The output of the `AdmissionControlWithWorkloadPartitioning` is a list of TPUs and the TPU units allocated on each TPU for the requested model. The extended scheduler uses this output to configure the weights for the respective LBS.

5 MICROEDGE DATA PLANE

MicroEdge's control plane elevates the TPU as a first-class citizen from a scheduling perspective, with the goal of ensuring that the scarce TPU resources will be fully utilized. It accomplishes this goal via granular allocation of TPU resources without dedicating a TPU exclusively for an application, and workload partitioning to fan out an application's successive requests to different TPUs.

To support these admission control mechanisms of the control plane, the data plane implements mirroring data plane mechanisms for seamless execution of the inference requests on the TPUs with minimal runtime overhead. Specifically, the data plane provides the ability to space and time share a given TPU across multiple applications via *TPU Service* (§ 5.1); relieves the application developer by providing client library dubbed *TPU Client* (§ 5.2); and facilitates workload partitioning for requests from a given application via *TPU load balancing service* (§ 5.3).

5.1 TPU Service

In MicroEdge, we allow sharing of a TPU's resources across independent application pipelines by exposing each TPU as a *service*. Each *tRPi* (*i.e.*, RPI endowed with a TPU) in MicroEdge runs a TPU Service, which is instantiated when the MicroEdge cluster is booted up.

The TPU Service *listens* for two kinds of incoming requests: *Load* and *Invoke*. When a TPU Service receives a *Load* request (from extended scheduler), it loads the specified machine learning model to the associated TPU's memory. When a TPU Service receives an *Invoke* request (from a TPU Client), it runs inference for the image frame received and sends the results back.

The TPU Service facilitates time sharing and space sharing, both of which are key to enhancing the TPU utilization:

- **Time Sharing.** Multiple applications can share a single TPU by sending their requests to the same TPU Service. While these requests will be executed serially in the order in which they are received, the admission control done by the extended scheduler ensures that the application SLOs will be met.
- **Space Sharing.** MicroEdge uses TPU's co-compiling feature to provide space sharing. The extended scheduler (§ 4) could co-compile models from different applications and load the composite models into the TPU memory so long as the TPU has sufficient memory to host the composite models. Loaded with co-compiled models, a single TPU service can serve different models without paying the overhead of swapping the models in/out between the host memory and the TPU memory.

5.2 TPU Client

As we mentioned earlier (§ 3.2), TPU Client is a Python library included with the application for issuing *Invoke* requests to the TPU Service (s) serving this application pod. Many ML models expect an input resolution that is smaller than the original image frame's resolution. Baking the TPU Client with the application ensures that the image resizing is done on the client side before being sent to the TPU Service (via the cluster interconnect); this is critical since the data movement overhead is significant on low-cost devices such as RPis (§ 6.4.2).

5.3 TPU Load Balancing Service

Each application pod has a load balancing component attached to it as shown in Fig. 3. At the time of initializing the application pod, the extended scheduler seeds the associated load balancing service (LBS) with the work partitioning weights (§ 4.3) so that the *Invoke* requests emanating from its TPU Client can be routed to the appropriate TPU Service (s). In MicroEdge, we choose to implement our own LBS instead of using K3s's default LBS [23]. K3s's default LBS does not offer the capability to send requests to specific TPUs, which is required for the correct functioning of the work-partitioning scheme enshrined in the extended scheduler. Our LBS forwards requests from the TPU Client to TPU Services using Weighted Round Robin (WRR) with Weight Fair Queuing (WFQ) spread [8] [32].

6 EVALUATION

The experimental study will answer the following research questions:

(1) How well does MicroEdge meet the *Service Level Objectives (SLOs)*, namely throughput and latency, for camera processing applications? How well does MicroEdge scale with the number of camera instances?
(2) How much gain in TPU utilization is achieved by MicroEdge due to TPU sharing and reduction in the cost of ownership in return?
(3) How well does MicroEdge work for real-world use cases, wherein a diverse mix of inference requests with varying life times are dynamically generated from camera streams?
(4) What are the overheads attributable to MicroEdge in the control plane and data plane?

An important metric for camera processing applications such as object tracking across geo-distributed cameras is *scalability*, *i.e.*, the ability of the system to cater to an increasing number of camera streams. Further, such applications show that a minimum frame rate is needed to meet the fidelity requirements (*i.e.*, low false positives and false negatives). Therefore, throughput is a critical SLO in MicroEdge.

Moreover, to achieve a low cost for camera processing in an Edge computing cluster, service providers should improve the utilization of hardware (*i.e.*, servers and accelerators) to reduce the total cost of ownership. Therefore, we also focus on TPUs' utilization in the evaluation of MicroEdge. To sum it up, for the evaluation study of MicroEdge, the metrics of interest are scalability, throughput/frame rate, and TPU utilization. The experimental setup for the performance evaluation is covered in § 6.1; § 6.2 reports on the scalability study using two exemplar applications; using a real-world function trace the versatility of MicroEdge to deliver on the metrics of interest is discussed in § 6.3; finally, the micro-measurements summarized in § 6.4 confirm that the overheads incurred due to MicroEdge are minimal.

6.1 Experiment Setup

Software Setup. We build the proposed multi-tenancy architecture by extending K3s's control plane and data plane. Specifically, we implement the data plane extensions (*i.e.*, TPU Service, LBS, and TPU Client) shown in Fig. 3 in Python. We implement the extended scheduler and Reclamation components shown in Fig. 3 in Go and package the Co-compiler as a service in Python. In the course of the evaluation, we use the detection ML model SSD MobileNet V2 and human body segmentation model BodyPix MobileNet V1 in § 6.2, and we use the classification model MobileNet V1 and segmentation model UNet V2 in § 6.3 to simulate a real-world workload.

Hardware Setup. We use 25 Raspberry Pi 4 Model B with the following specifications: Quad-core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz, 8GB LPDDR4-3200 SDRAM. Additionally, we use six Google Edge TPU ML accelerator co-processors. By having fewer TPUs than the number of RPis, we demonstrate the cost-effectiveness of MicroEdge. Depending on the specifics of the experiments, we will use either the entire MicroEdge cluster or a subset of the hardware in the cluster.

6.2 Scalability Study

This experiment sets out to answer the first two research questions posed at the beginning of § 6.

Applications. We use two exemplar situation awareness applications. The first one dubbed Coral-Pie generates space-time tracks of vehicles at video ingestion time using a distributed camera network [38]. The bare-metal implementation of this application used as the baseline for our evaluation study dedicates two RPis and one TPU for each camera stream to sustain the 15 FPS SLO. The first RPi runs pre-processing and inferences on the TPU to detect vehicles from the camera's field of view, and the second RPi re-identifies vehicles reported by upstream cameras.

In fact, the two RPis process their workloads independently in a pipelined fashion. Therefore, in principle we can study the workload on either RPi separately as long as each RPi completes its processing within the time available for per-frame processing. Since the focus of our evaluation is on TPU utilization, the workload we use for the evaluation is confined to the vehicle detection pipeline that runs on the first RPi with the attached TPU. The second application is Google Coral BodyPix [4], which does real-time person segmentation. For this application, the bare-metal baseline to compare against MicroEdge uses two TPUs attached to each RPi host. The segmentation model used by this application has a TPU unit > 1 at 15 FPS, thus requiring two TPUs to maintain the frame rate SLO. Throughout our evaluation we use a configuration where TPUs are dedicated to a single application as the baseline. To the best of our knowledge, this baseline is itself the first of its kind incorporating TPUs in a multi-tenant edge cluster. The study quantifies the utilization improvements achievable via the mechanisms such as TPU multiplexing proposed in this paper compared to this baseline.

Dataset. For the Coral-Pie application, we evaluate MicroEdge using a video file that contains 1000 image frames recorded from a security camera capturing the movement of vehicles in a campus environment. The time it takes a vehicle to enter and leave the FOV of a camera is around 10 seconds. At 15 FPS, 1000 image frames amount to 67 seconds of video, which allows observing several vehicles traversing through the FOV of a camera. Further, we give the same set of frames "time shifted" (so that we know the ground truth for vehicle re-identification) to downstream cameras. So cumulatively, the dataset used in the evaluation is equivalent to 20,000 camera frames. For the BodyPix application, we take 1000 images from a 3D people dataset [31].

Workload. With the cameras fed with the video datasets described above, we use the detection model SSD MobileNet V2 for Coral-Pie, and the segmentation model BodyPix MobileNet V1 for BodyPix as the ML workloads to send to the TPUs. We set the frame rate of each camera to the industry-recommended [16] 15 FPS. Since the BodyPix segmentation model requires > 1 TPU unit at 15 FPS, the baseline needs 2 TPUs for each camera instance, sending alternate frames to each TPU to meet the frame rate SLO. For the Coral-Pie application which requires a TPU unit < 1 for the ML model, we use the two MicroEdge scheduling variants with and without workload partitioning. For the BodyPix application which requires a TPU unit > 1, we only use the workload partitioning scheduling variant. Using the setup described above, we run the two applications separately to get deterministic workloads. In the experiments, we increase the number of cameras until it reaches the maximum capacity for each configuration.

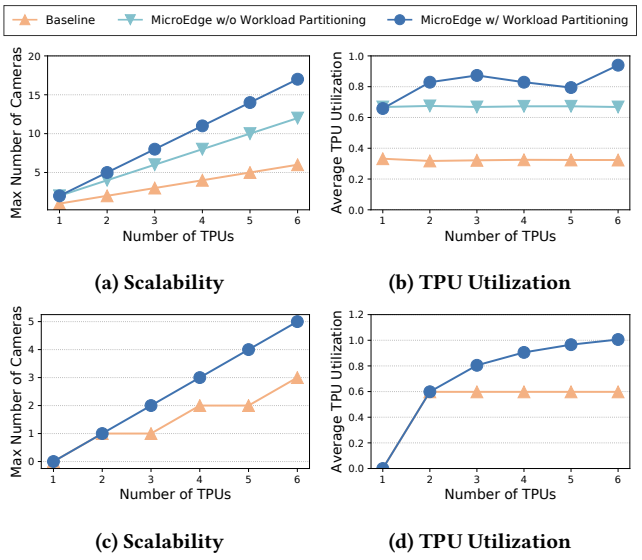


Figure 5: Scalability of MicroEdge. Fig. 5a and Fig. 5b show the results for the Coral-Pie application, while Fig. 5c and Fig. 5d show the results for the BodyPix application. Fig. 5a and Fig. 5c compare the scalability of MicroEdge against the baseline with 15-FPS camera instances. Fig. 5b and Fig. 5d show the corresponding average utilization of all the TPUs in the cluster.

Scalability Results. The detection ML model used by Coral-Pie needs 0.35 TPU units, while the segmentation ML model used by BodyPix needs 1.2 TPU units. The bare-metal baselines which dedicate TPUs for each camera stream cannot exploit fractional TPU resources. MicroEdge scheduling variants on the other hand can fully exploit such fractional availability of TPU resources via TPU sharing. The scalability of MicroEdge compared to the baselines is evident from Fig. 5a and Fig. 5c. Succinctly put, for the same number of TPUs, MicroEdge can support more number of camera streams compared to the baselines for both computationally light (Fig. 5a), and computationally heavy (Fig. 5c) models. For *e.g.*, with workload partitioning (Fig. 5a), MicroEdge can support up to 2.8× cameras compared to the baseline with 6 TPUs for the Coral-Pie application. Comparing the results with and without workload partitioning for the Coral-Pie application (Fig. 5a) underscores the importance of the workload partitioning scheduling variant of MicroEdge for enhancing the scalability even further.

TPU Utilization Results. Because of MicroEdge's granular scheduling of TPU resources that aids TPU sharing across independent application pipelines, its scheduling variants achieve higher TPU utilization compared to the baselines. With reference to Fig. 5b, the TPU utilization could be as low as 33% for the baseline. On the other hand, MicroEdge even without workload partitioning (Fig. 5b) achieves a TPU utilization of up to 70%. The TPU utilization reaches almost 100% with workload partitioning as we scale up the cluster size (Fig. 5b and Fig. 5d). The scalability of MicroEdge

	# TPUs	# RPis	Total Cost
Baseline	17	17	$2550
MicroEdge w/o W.P.	8	17	$1875
MicroEdge w/ W.P.	6	17	$1725

Table 1: Cost comparison between the baseline and MicroEdge variants to support 17 Coral Pie camera instances.

also has a direct impact on the cost of ownership. As we can see in Table 1, to support 17 cameras for Coral-Pie, MicroEdge with workload partitioning reduces the cost of ownership by 33%.[4]

Throughput Results. We conducted experiments to verify if virtualization or K3s extensions in the data plane add any significant overhead to camera processing. Compared to running the baseline on bare metal, the virtualized MicroEdge cluster does have inherent overhead since the client applications and the other entities in Fig. 3 are containerized for performance isolation. However, by proof of construction we show that this overhead is minimal. The measured throughput results of the MicroEdge scheduling variants are the same as the baselines. Of course, compared to the baselines which run on bare metal, MicroEdge with containerization and the entities in the data plane of K3s do add additional latency for camera processing. We discuss the sources of this latency in § 6.4. Suffice it to say here that despite the additional latency incurred per frame, at the application level, MicroEdge is able to sustain the needed throughput for typical camera processing applications.

6.3 Real World Workload Study

In a realistic workload, MicroEdge cluster should be able to simultaneously serve requests for multiple models. Additionally, applications may be created and removed in an unpredictable pattern. This section reports on experiments with a larger cluster and uses a realistic workload to simulate clients 'coming and going'.

Workload Setup. The Microsoft Azure Functions (MAF) [36] trace gives the frequency of invocations of user-created serverless functions gathered over 2 weeks. In our evaluation, we ascribe each invocation to a specific camera stream. That is, the number of invocations of a specific function in the trace becomes the number of camera streams in our study. To fit the limited capacity of the MicroEdge cluster we downsize the number of invocations so that they would not exceed MicroEdge's system capacity. Despite this modification, we retain the diversity of the functions (e.g., duration of each function, function periodicity).

For this experiment, we simulate a scenario where MicroEdge simultaneously serves three types of camera streams corresponding to three models as described in § 6.1. We also associate three different attributes, respectively, to the three models (derived from MAF): one which assumes 24 x 7 processing, one which shows a sparse invocation, and one with bursty requests.

We run the trace through four different configurations of MicroEdge and test our workload partitioning and co-compiling features. Each of the configurations either have both features enabled, only one of the features enabled, or none of the features enabled. The baseline for this study is identical to that of § 6.2 where each TPU is dedicated to a single camera.

TPU Utilization Results. In this part, we analyze the total TPU utilization of the different MicroEdge configurations and compare them with the baseline. Fig. 6a shows the average TPU utilization of each of the configurations per minute. The most visible observation is that the baseline utilization is fixed at a low level, whereas MicroEdge shows a high utilization above 0.7 and reaching 1 at certain times. The TPU utilization of baseline is capped as expected since each TPU can handle requests from only one camera. From the results. we can see that using both workload partitioning and co-compiling yields the highest level of utilization. The fluctuation in the utilization of MicroEdge for the different configurations is due to the variance of the workload imposed upon the cluster. Note that the separation between each configuration is not always uniform, once again attributable to the workload variance.

Throughput Results. Fig. 6b shows the number of cameras that MicroEdge can serve for the different configurations. Similar to Fig. 6a, we can see that MicroEdge with both workload partitioning and co-compiling shows the best throughput performance since both features exploit the marginal resource of each TPU to increase utilization. When used alone, co-compiling supports a higher number of cameras compared to workload partitioning in general.

Workload partitioning farms out a single camera's requests to multiple TPUs, whereas co-compiling allows a single TPU to host multiple models which in turn cater to the inference requests from multiple cameras, thus allowing a given TPU to serve more number of cameras.

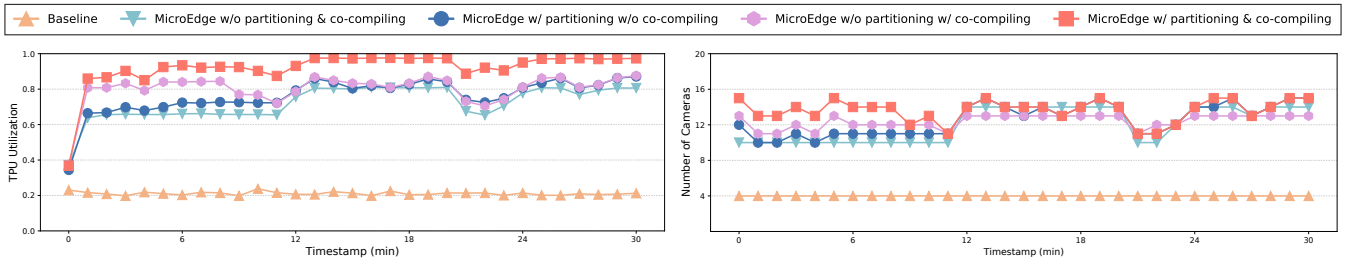
To fully understand the benefits of co-compilation and workload partitioning, we would need to run a much larger configuration of the workload on a larger cluster. Such a study would show a stronger separation in the results for the different configurations.

6.4 Micro Measurements

6.4.1 One-time Admission Control Overhead. This benchmark quantifies the increase in admission control latency over and above K3s due to the additional work done by the control plane of MicroEdge. The additional work done by the control plane includes camera stream admission, node selection, (optional) co-compilation, and TPU load balancing before the container is launched. As can be seen in Fig. 7a, the additional latency overhead is around 10% for launching a new camera instance. When the camera runs a new model that needs to be co-compiled, even though the latency overhead of admission control has a larger variance, the average value does not increase because the co-compilation runs on a different process in parallel with the extended scheduler. It should be noted that this is a one-time control plane overhead for launching the application and is not in the critical path of application processing. Given that these applications are long-running the one-time additional control plane cost is acceptable.

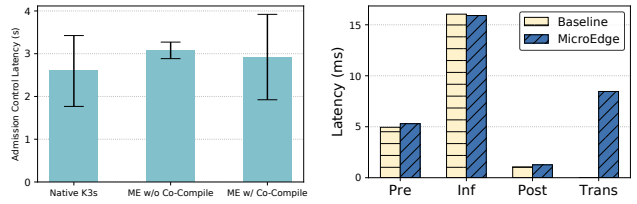
6.4.2 Inference Latency Breakdown. This microbenchmark is designed to break down the end-to-end latency for each inference in MicroEdge. There are four steps in each request. ❶ Pre-Processing: Examples include resizing the raw image to the resolution that fits the ML model. ❷ Transmission: TPU Client sends pre-processed

[4] We do not include the cost of the remote server that hosts the control plane. This cost can easily be amortized over multiple MicroEdge clusters deployed across a metropolitan area.



(a) TPU Utilization **(b) Number of Cameras MicroEdge Can Handle**

Figure 6: MicroEdge Performance Under Trace Workload– Fig. 6a shows how the TPU utilization changes for different configurations of MicroEdge as time passes. Fig. 6b shows the number of camera clients that different configurations of MicroEdge can handle under a fluctuating workload.



(a) Admission Control Overhead **(b) *Invoke*'s Latency Breakdown**

Figure 7: Latency Measurements. Fig. 7a compares the admission control overhead of native K3s and MicroEdge. Fig. 7b shows a breakdown of the latency involved in *Invoke* call for Coral-Pie baseline and MicroEdge along four component dimensions: Preprocessing, Inference, Post-processing, and Transmission.

image to its assigned TPU Service. ❸ Inference: the TPU Service loads the image on TPU and runs inference. ❹ Post-Processing: the result is returned to the application for post-processing.

Fig. 7b shows the latency breakdown of the Coral-Pie application [38] used in § 6.2. The baseline does not need step 4 since the TPU is dedicated to the camera (and hence collocated with the respective RPi).

As is evident from Fig. 7b, the dominant cost of the latency is the extra data transmission incurred to transport the image from the application container to the TPU Service (around 8 ms). The total end-to-end latency for an inference request adds up to only 31 ms. A model that can satisfy the FPS target in a baseline K3s configuration would most likely be able to satisfy the FPS target in MicroEdge. The goal of MicroEdge is to increase resource utilization while respecting the latency SLAs of the applications. We honor latency constraints from applications, but do not strive to minimize the latency for each request. In fact, it would not benefit the application pipeline to do such latency reduction. A common camera application SLA of 15 FPS gives a latency budget of 67 ms between frames. Even if the entire application pipeline finishes the execution of one frame in 20 ms, the next frame will arrive only after 47 ms due to the periodicity of frame arrival. This is the reason the focus of this work is to increase the utilization while meeting the application SLA. In general, we observe that a majority of models also have similar input sizes (e.g., 230 x 230, 300 x 300). The transmission latency overhead is consistent for different models used in the evaluation because images are resized by the TPU Client to the model-specific size before transmission to the TPU Service. It justifies that MicroEdge is able to meet the latency requirements of most camera processing applications which require 15 FPS [16].

7 RELATED WORK

GPU Schedulers at Edge. [18] places camera stream processing pipelines on a cluster of NVIDIA Jetson Nano boards. They place individual pipeline components instead of placing a monolithic pipeline to provide mechanisms to establish communication between components. This allows different pipelines to share overlapping components and heavy workload pipelines to span across multiple hardware nodes. However, their work does not support automated scheduling and requires human intervention when placing components. They also do not deal with possible hardware fragmentation. On the contrary, MicroEdge provides a control plane that provides automatic, fine-grained TPU scheduling which actively minimizes the TPU fragmentation. MicroEdge also provides performance isolation between different applications, which is critical in serving an undefined audience.

DeepRT [39] is a GPU scheduler for a multi-tenant edge server that aims to provide latency guarantee to the inference requests while maintaining high overall system throughput. DeepRT batches data from different requests as much as possible to boost the system throughput. Meanwhile, the Admission Control Module in DeepRT performs a simulation-based schedulability test to decide whether a pending request's latency guarantee can be satisfied. In contrast to the GPU, the TPU's RAM is much more limited, making batch processing of images infeasible in MicroEdge.

GPU Schedulers at Cloud. Clipper [7] takes only the image as input and automatically chooses the model that best serves the request. INFaas [33] expands the search space to variants of a model and supports autoscaling of resources based on the workload. Clockwork [14] uses a central queue and limits variances in the inference pipeline to achieve predictable tail latencies.

While these works also abstract hardware and handle inference at scale, there are several differences with MicroEdge. These works

handle completely unpredictable workloads and must allocate resources at every inference. On the contrary, MicroEdge targets camera streams that generate frames at predictable intervals. This allows MicroEdge to allocate resources at camera stream granularity. Another difference comes from the fact that resources are scarce on the Edge compared to the Cloud and must be used wisely. Because of this, MicroEdge's key focus is on minimizing resource fragmentation and serving as many clients as possible with limited hardware as long as the SLO is satisfied.

Contemporaneous with our work, Cho, *et al.* [5] have proposed a framework for deploying ML workers on a heterogeneous GPU cluster based on SLO requirements which bears similarity to some of the ideas in MicroEdge. However, they do not discuss their resource allocation algorithm in detail and do not provide an extensive evaluation on the scalability of their framework.

Systems in the Industry. NVIDIA MPS[26] uses the Hyper-Q capability of GPUs to allow CUDA kernels to be processed concurrently on the same GPU, which can benefit performance when a single application process underutilizes the GPU. TensorFlow-Serving [29] and Triton Inference Server [28] are open-source industrial inference systems for GPUs. These two inference systems choose a serverless design wherein scheduling decisions are made per function call during the runtime.

In addition, Amazon SageMaker [1] and Vertex AI [13] from Google are two closed-source model serving systems in the industry. Meanwhile, TPUs are much younger, and similar industry support does not exist.

8 CONCLUSION

MicroEdge is a low-cost edge cluster comprising RPis and TPUs offering computational resources for geo-local camera processing applications. Camera streams generate image frames at a fixed interval, and the discrepancy between the interval and inference time leads to fragmentation if computational resources are dedicated to applications. Hence, MicroEdge features a performant multi-tenancy architecture to share and orchestrate resources (*i.e.*, CPU, TPU, and memory) among geo-local camera applications. MicroEdge proposes novel mechanisms all aimed at ensuring that the precious TPU resources are fully utilized. The mechanisms embodied in the extended scheduler that extends the K3s native scheduler include time-sharing individual TPUs across multiple camera streams, partitioning the requests from a given camera stream across multiple TPUs, and space-sharing a TPU for interleaving inference requests from camera streams using different models via co-compilation. MicroEdge implements admission control by extending K3s's control plane and data plane, allocating resources upon creating the application pod, and reclaiming the resources upon application termination. Extensive evaluation studies are carried out to showcase the performance advantages of MicroEdge for camera processing applications. As a camera processing cluster at the edge, MicroEdge offers several avenues for future work. Some examples include data plane optimization for pipelines that involve multiple models, automated model partitioning, and support for failure recovery.

ACKNOWLEDGMENTS

We thank our anonymous reviewers for their insightful feedback and suggestions, which substantially improved the content of this paper. This work was funded in part by NSF CNS-2008368, Cisco, and a gift from Microsoft Corp.

REFERENCES

[1] Amazon. 2022. *What Is Amazon SageMaker?* Retrieved October 3, 2022 from https://docs.aws.amazon.com/sagemaker/latest/dg/whatis.html
[2] K3s Project Authors. 2022. *K3s: Lightweight Kubernetes.* Retrieved October 3, 2022 from https://k3s.io/
[3] Johan Barthélemy, Nicolas Verstaevel, Hugh Forehead, and Pascal Perez. 2019. Edge-Computing Video Analytics for Real-Time Traffic Monitoring in a Smart City. *Sensors* 19, 9, Article 2048 (May 2019), 23 pages. https://doi.org/10.3390/s19092048
[4] Michael Brooks, Naveen-Dodda, and Peter Malkin. 2021. *Google Coral BodyPix.* Retrieved October 3, 2022 from https://github.com/google-coral/project-bodypix.git
[5] Junguk Cho, Diman Zad Tootaghaj, Lianjie Cao, and Puneet Sharma. 2022. SLA-Driven ML Inference Framework For Clouds With Heterogeneous Accelerators. In *Proceedings of the 5th Conference on Machine Learning and Systems* (Santa Clara, California, August 29 - September 1, 2022) *(MLSys '22)*. 20–32. https://proceedings.mlsys.org/paper/2022/file/0777d5c17d4066b82ab86dff8a46af6f-Paper.pdf
[6] Wikipedia contributors. 2022. *Bin packing problem.* Retrieved October 3, 2022 from https://en.wikipedia.org/w/index.php?title=Bin_packing_problem
[7] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation* (Boston, Massachusetts, March 27 - 29, 2017) *(NSDI '17)*. USENIX, Berkeley, CA, USA, 613 – 627. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw
[8] Alan Demers, Srinivasan Keshav, and Scott Shenker. 1989. Analysis and Simulation of a Fair Queueing Algorithm. *SIGCOMM Comput. Commun. Rev.* 19, 4 (Aug 1989), 1 – 12. https://doi.org/10.1145/75247.75248
[9] The Raspberry Pi Foundation. 2022. *Raspberry Pi 4 Model B.* Retrieved October 3, 2022 from https://www.raspberrypi.com/products/raspberry-pi-4-model-b/
[10] Google. 2020. *Co-compiling multiple models.* Retrieved October 3, 2022 from https://coral.ai/docs/edgetpu/compiler/#co-compiling-multiple-models
[11] Google. 2020. *Parameter data caching.* Retrieved October 3, 2022 from https://coral.ai/docs/edgetpu/compiler/#parameter-data-caching
[12] Google. 2020. *What is the Edge TPU?* Retrieved October 3, 2022 from https://coral.ai/docs/edgetpu/faq/
[13] Google. 2022. *Vertex AI.* Retrieved October 3, 2022 from https://cloud.google.com/vertex-ai
[14] 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 *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation* (Virtual, November 4 - 6, 2020) *(OSDI '20)*. USENIX, Berkeley, CA, USA, 443 – 462. https://www.usenix.org/conference/osdi20/presentation/gujarati
[15] Intel. 2022. *Intel Neural Compute Stick 2.* Retrieved October 3, 2022 from https://software.intel.com/content/www/us/en/develop/hardware/neural-compute-stick.html
[16] IPVM. 2021. *Average Frame Rate Video Surveillance Statistics 2021.* Retrieved October 3, 2022 from https://ipvm.com/reports/average-frame-rate-video-surveillance-2021
[17] Samvit Jain, Xun Zhang, Yuhao Zhou, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, Paramvir Bahl, and Joseph Gonzalez. 2020. Spatula: Efficient cross-camera video analytics on large camera networks. In *Proceedings of the 5th IEEE/ACM Symposium on Edge Computing* (Virtual, November 11 - 13, 2020) *(SEC '20)*. ACM, New York, NY, USA, 110–124. https://doi.org/10.1109/SEC50012.2020.00016
[18] Si Young Jang, Boyan Kostadinov, and Dongman Lee. 2021. Microservice-based Edge Device Architecture for Video Analytics. In *Proceedings of the 6th IEEE/ACM Symposium on Edge Computing* (San Jose, California, December 14 - 17, 2021) *(SEC '21)*. ACM, New York, NY, USA, 165 – 177. https://doi.org/10.1145/3453142.3491283
[19] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2017. NoScope: Optimizing Neural Network Queries over Video at Scale. In *Proceedings of the 43rd International Conference on Very Large Data Bases* (Munich, Germany, Auguest 28 - September 1, 2017) *(VLDB '17)*. VLDB Endowment, Los Angeles, CA, USA, 1586 – 1597. https://doi.org/10.14778/3137628.3137664
[20] Kubernetes. 2022. *Assigning Pods to Nodes.* Retrieved October 3, 2022 from https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/

[21] Kubernetes. 2022. *Managing Resources for Containers.* Retrieved October 3, 2022 from https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/

[22] Kubernetes. 2022. *Production-Grade Container Orchestration.* Retrieved October 3, 2022 from https://kubernetes.io/

[23] Kubernetes. 2022. *Service.* Retrieved October 3, 2022 from https://kubernetes.io/docs/concepts/services-networking/service/

[24] NVIDIA. 2020. *CUDA Multi-process Service.* Retrieved October 3, 2022 from https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf

[25] NVIDIA. 2022. *Jetson Nano Developer Kit.* Retrieved October 3, 2022 from https://developer.nvidia.com/embedded/jetson-nano-developer-kit

[26] NVIDIA. 2022. *Multi-Process Service.* Retrieved October 3, 2022 from https://docs.nvidia.com/pdf/CUDA_Multi_Process_Service_Overview.pdf

[27] NVIDIA. 2022. *NVIDIA Container Toolkit.* Retrieved October 3, 2022 from https://github.com/NVIDIA/nvidia-docker

[28] NVIDIA. 2022. *NVIDIA Triton Inference Server.* Retrieved October 3, 2022 from https://developer.nvidia.com/nvidia-triton-inference-server

[29] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. 2017. Tensorflow-serving: Flexible, high-performance ml serving. In *Workshop on ML Systems at NIPS 2017* (Long Beach, California, December 8, 2017). 8 pages. http://learningsys.org/nips17/assets/papers/paper_1.pdf

[30] George Papandreou, Tyler Zhu, Liang-Chieh Chen, Spyros Gidaris, Jonathan Tompson, and Kevin Murphy. 2018. PersonLab: Person Pose Estimation and Instance Segmentation with a Bottom-Up, Part-Based, Geometric Embedding Model. In *Proceedings of the 15th European Conference on Computer Vision* (Munich, Germany, September 8 - 14, 2018) *(ECCV '18)*. Springer, Cham, Switzerland, 282 – 299. https://doi.org/10.1007/978-3-030-01264-9_17

[31] Albert Pumarola, Jordi Sanchez, Gary P. T. Choi, Alberto Sanfeliu, and Francesc Moreno. 2019. 3DPeople: Modeling the Geometry of Dressed Humans. In *Proceedings of the 17th IEEE/CVF International Conference on Computer Vision* (Seoul, Korea, October 27 - November 2, 2019) *(ICCV '19)*. IEEE, New York, NY, USA, 2242–2251. https://doi.org/10.1109/ICCV.2019.00233

[32] Python. 2020. *Python roundrobin 0.0.2.* Retrieved December 12, 2021 from https://pypi.org/project/roundrobin/

[33] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. 2021. INFaaS: Automated Model-less Inference Serving. In *Proceedings of the 2021 USENIX Annual Technical Conference* (Virtual, July 14 - 16, 2021) *(ATC '21)*. USENIX, Berkeley, CA, USA, 397–411. https://www.usenix.org/conference/atc21/presentation/romero

[34] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *Proceedings of the 2018 IEEE Conference on Computer Vision and Pattern Recognition* (Salt Lake City, Utah, June 18 - 22, 2018) *(CVPR '18)*. IEEE, New York, NY, USA, 4510 – 4520. https://doi.org/10.1109/CVPR.2018.00474

[35] Mahadev Satyanarayanan. 2017. The Emergence of Edge Computing. *Computer* 50, 1 (Jan 2017), 30–39. https://doi.org/10.1109/MC.2017.9

[36] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *Proceedings of the 2020 USENIX Annual Technical Conference* (Virtual, July 15 - 17, 2020) *(ATC '20)*. USENIX, Berkeley, CA, USA, 205–218. https://www.usenix.org/conference/atc20/presentation/shahrad

[37] TensorFlow. 2022. *TensorFlow Hub.* Retrieved October 3, 2022 from https://tfhub.dev/

[38] Zhuangdi Xu, Harshil Shah, and Umakishore Ramachandran. 2020. Coral-Pie: A Geo-Distributed Edge-compute Solution for Space-Time Vehicle Tracking. In *Proceedings of the 2020 ACM/IFIP Middleware* (Delft, the Netherlands, December 7 - 11, 2020) *(Middleware '20)*. ACM, New York, NY, USA, 400 – 414. https://doi.org/10.1145/3423211.3425686

[39] Zhe Yang, Klara Nahrstedt, Hongpeng Guo, and Qian Zhou. 2021. DeepRT: A Soft Real Time Scheduler for Computer Vision Applications on the Edge. In *Proceedings of the 6th IEEE/ACM Symposium on Edge Computing* (San Jose, California, December 14 - 17, 2021) *(SEC '21)*. ACM, New York, NY, USA, 271 – 284. https://doi.org/10.1145/3453142.3491278

[40] Juheon Yi and Youngki Lee. 2020. Heimdall: Mobile GPU Coordination Platform for Augmented Reality Applications. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking* (London, United Kingdom, September 21 - 25, 2020) *(MobiCom '20)*. ACM, New York, NY, USA, 462 – 475. https://doi.org/10.1145/3372224.3419192

[41] Mingming Zhang, Chaochao Chen, Tianyu Wo, Tao Xie, Md. Zakirul Alam Bhuiyan, and Xuelian Lin. 2017. SafeDrive: Online Driving Anomaly Detection From Large-Scale Vehicle Data. *IEEE Transactions on Industrial Informatics* 13 (Dec 2017), 2087–2096.