

IRIS: Interference and Resource Aware Predictive Orchestration for ML Inference Serving

Aggelos Ferikoglou

National Technical University of Athens

Athens, Greece

aferikoglou@microlab.ntua.gr

Panos Chrysomeris

National Technical University of Athens

Athens, Greece

pchrysomeris@microlab.ntua.gr

Achilleas Tzenetopoulos

National Technical University of Athens

Athens, Greece

atzenetopoulos@microlab.ntua.gr

Manolis Katsaragakis

National Technical University of Athens

Athens, Greece

mkatsaragakis@microlab.ntua.gr

Dimosthenis Masouros

National Technical University of Athens

Athens, Greece

demo.masouros@microlab.ntua.gr

Dimitrios Soudris

National Technical University of Athens

Athens, Greece

dsoudris@microlab.ntua.gr

Abstract—Over the last years, the ever-growing number of Machine Learning (ML) and Artificial Intelligence (AI) applications deployed in the Cloud has led to high demands on the computing resources required for efficient processing. Multiple users deploy multiple applications on the same server node to maximize Quality of Service (QoS); however, this leads to increased interference. In addition, Cloud providers aim to minimize their operating costs by efficiently utilizing the available resources. These conflicting optimization goals form a complex paradigm where efficient scheduling is required.

In this work, we present IRIS, an interference- and resource-aware predictive inference scheduling framework for ML inference serving in the cloud. We target the multi-objective problem of QoS maximization with effective CPU utilization based on Queries per Second (QPS) predictions by proposing a model-less ML-based solution and integrating it into the Kubernetes platform. Our approach is evaluated over real hardware infrastructure and a set of ML applications. Our experimental analysis shows that under various QoS constraints, the model-specific interference-aware scheduler violates QoS constraints less frequently by achieving $1.8\times$ fewer violations, on average, compared to over-provisioning and $3.1\times$ fewer violations compared to under-provisioning, through efficient exploitation of available CPU resources. The model-less feature is able to cause, on average, $1.5\times$ fewer violations compared to the model-specific scheduler, while further reducing the average CPU utilization by $\approx 30\%$.

Index Terms—Cloud, Machine Learning, Inference, Scheduling, Interference-aware, Resource-aware, Model-less

I. INTRODUCTION

Today, the number of applications that utilize Artificial Intelligence (AI) is increasing in high pace. Deep Learning (DL) [1] models that provide inference over huge datasets appear over several alternative domains, with healthcare [2], finance [3] and transportation [4] being just a few representative examples. The rise of such applications, in addition to the ever-increasing complexity of the deployed models, has led to increased demands in terms of computation, memory and storage resources. Especially for modern deep learning

models, the required resources may skyrocket, thus making it prohibitive for the end-users to support inference. For example, the ResNet-50 architecture [5] has over 25 million parameters, while the latest popular models, such as GPT-3 [6], require more than 175 billion parameters.

Cloud comes to alleviate this resource wall challenge, by taking a significant amount of the computational burden, providing scalable, flexible and cost-effective solutions. As a matter of fact, inference accounts for more than 90% of total infrastructure costs within AWS [7], while Facebook runs online inference tasks tens-of trillions of times per day [8]. To ease the deployment of inference workloads, Cloud providers offer Machine Learning solutions as services over their infrastructure (MLaaS), with typical examples including IBM Watson [9], Google’s Vertex AI [10], Amazon SageMaker [11] and Azure Machine Learning [12]. In the MLaaS paradigm, end-users are able to upload their pre-trained models and expose them as web-services through specific APIs. Typically, these models are accompanied by throughput- or latency-oriented Quality of Service (QoS) requirements (e.g., “model must serve at least 60 requests per second”) or bound by Service Level Objectives (SLOs) (e.g., “at least 99% of the requests must be completed within 500ms”) [8], [13], [14]. To achieve these requirements, providers tend to be conservative and over-provision resources allocated for such services [13], in order to keep up with the highly fluctuating load over the day, which, however, leads to resource under-utilization and increased operational costs [15], [16].

As such, a key challenge for Cloud providers becomes to guarantee the aforementioned requirements, while also maximizing the resource efficiency of their infrastructure. However, this co-optimization objective is extremely challenging, due to: *i) Resource Interference*: Providers tend to co-locate applications on the same physical servers to increase the resource utilization of their infrastructures [15], [17], [18]. However, this leads to interference in the shared resources of the systems, which in turn is translated to high performance variability [16]. Especially for latency-critical applications,

even small amounts of interference can lead to significant QoS violations [19], [20]; *ii) Incoming Requests Variability:* Incoming requests can vary in their timing, frequency, and workload characteristics, making it challenging to provide consistent and predictable performance [21], [22]; *iii) Diverse QoSs/SLAs:* While applications from different domains apparently use the same models for their tasks (e.g., ResNet for image classification), they can introduce different QoS requirements. For example, automotive applications require millisecond-scale inference serving [23], while agricultural ones can be more tolerant to performance variability [24]. *iv) Model Variants:* There exist a vast amount of different DNN architectures (e.g., ResNet, MobileNet), or different versions of the same architecture (e.g., quantized MobileNet), available for performing the same task (e.g., image classification). The abundance of different frameworks such as TensorFlow, PyTorch, ONNX Runtime, etc. represents another degree of freedom, as the exact model can be implemented on different backends. These variants trade performance for resource requirements and/or accuracy and vice-versa.

Aiming to overcome these challenges, efficient scheduling of ML inference serving systems deployed on the Cloud is required. However, sufficient scheduling is a quite challenging process, as a deep understanding of the application architecture, workload patterns, and Cloud infrastructure capabilities should be taken into account [25]. Cloud application scheduling has already been addressed and various solutions have been proposed [26]. Model-less approaches [27], i.e., scheduling frameworks that do not rely on pre-defined models, are becoming more and more famous in dynamic and unpredictable environments, where non-linear behavior is observed. Model-less scheduling usually relies on ML techniques, as they can rapidly adapt to varying conditions and real-time decision making, based on the current state and input workload.

In this work, we present IRIS, an interference and resource-aware, predictive scheduling framework for ML inference serving engines. The optimization objective of IRIS is to maximize QoS requirements, under efficient CPU utilization. IRIS supports both model-specific and model-less approaches, where the former allows end-users to directly specify the architecture of their ML serving system and the latter receives higher-level descriptions (e.g., image classification) and automatically determines the optimal inference serving solution. IRIS employs ML techniques predict the Queries per Second (QPS), based on the current load and resource state, and dynamically re-provisions CPU allocations to minimize resource utilization of the underlying system while also satisfying user-defined QoS requirements. The novel contributions of this work are the following:

- We propose a novel interference and resource-aware, predictive scheduling framework for ML inference engines, solving the multi-objective problem of QoS maximization with effective CPU utilization based on QPS prediction.
- We integrate a model-less approach into our scheduling framework which navigates the trade-off space of diverse ML model-variants for a specific inference task.

- We integrate and evaluate our solution with Kubernetes framework, showing that our scheduler is able to cause, on average, $1.5\times$ fewer violations compared to a model-specific interference-aware scheduler, while further reducing the average CPU utilization 30%, on average.

The rest of this paper is organized as follows. Section II presents an overview of related work, while in Section III we provide an extensive characterization of the target ML engines. In Section IV we present IRIS' architecture and in Section V the experimental evaluation, analysis and discussion is performed. Finally, Section VI concludes this work.

II. RELATED WORK

Inference Serving: One of the first approaches for inference serving was TensorFlow Serving [28], a flexible, high-performance serving system for machine learning models designed for production environments. From an industrial standpoint, Nvidia's AI platform offers the Triton Inference Server [29], which uses GPU inference serving while supporting CPU models, although requiring static configuration of the model instance. Towards this direction, several works with different approaches have been conducted, aiming to address the problem of efficient resource orchestration and optimization for ML inference serving systems. Adaptive and pre-defined batching techniques [30]–[33] have been introduced aiming to support ML inference, while auto-scaling approaches are also considered [32], [34], [35], [35], [36]. Furthermore, aiming to support efficient ML inference, server-less approaches have been considered [33], [37], while ML-based and predictive solutions for load request and resource utilization have been widely utilized [13], [14], [38]–[43], [43], [44], such as reinforcement learning-based solutions [43]. Aiming to provide lightweight decision making, the approach of model-less decision making has also been considered as an alternative [27], [45].

Interference-aware Scheduling: A major ability of a Cloud serving system is to operate efficiently under various interference levels. Several existing works aim to manage the complications caused by interference. Authors of [46] investigate the optimization of resource utilization through the awareness of the scheduler, based on past decision making. Moreover, a contention-aware scheduling approach to mitigate conflicts over shared resources has also been considered [47]. In [48], interference and QoS degradation models are utilized, by identifying the co-location of pre-characterized workloads, to improve data center utilization over interference. Resource partitioning techniques that aim to satisfy the QoS requirements have been examined [49], while in [17], [50], interference-aware scheduling on CPU servers, based on load predicting, is proposed. Finally, the authors of [51] consider a modular framework, aiming to balance incoming workload, based on low-level metrics monitoring.

Although various approaches to inference serving and interference-aware scheduling have been investigated in research, to the best of our knowledge, no study has yet combined the model-less approach with the interference- and

resource-aware feature. Our inference serving system uses different model-variants as well as vertical and horizontal scaling to counteract the effects of destructive interference, caused by other applications in the inference system. Through this approach, our methodology is able to meet user-defined QoS requirements and efficiently utilise the available CPU resources.

III. CHARACTERIZING INFERENCE SERVING

In this section, first, we provide a detailed presentation of our hardware and software setup, and also describe the inference serving benchmarks used in this paper. Then, we analyze the impact of resource availability on the performance of different popular deep learning models, commonly used for image processing tasks, while, on the same time, we unveil important insights concerning the sensitivity of the different models, as well as the different backend libraries, with respect to these two aspects, and we identify certain bottlenecks that relate to each case. The characterization process follows a Q&A approach, where the purpose of each question is to highlight important insights regarding different performance-related aspects.

A. Inference Serving Testbed

Hardware & Software Infrastructure: Our experiments were performed on a dual-socket Intel® Xeon® Gold 6138 (@2.0GHz) high-end server, equipped with 126 GB of DRAM memory. On the physical machine, we set up two virtual machines to serve as the master (4 vCPUs, 8 GB RAM) and worker (8 vCPUs, 16 GB RAM) nodes of our cluster, using KVM as the hypervisor. On top of the VMs, we deploy the Kubernetes container orchestrator (v1.23) in combination with Docker (v20.10), which is the most common method for deploying Cloud clusters at scale.

Inference Engine Workloads: For the purposes of this work, we use the object detection and image classification tasks derived from the MLPerf Inference benchmark suite [52]. Table I illustrates the MLPerf inference engines and model variant examined, with their corresponding accuracy. Each MLPerf inference container instance consists of two main components, *a)* the inference engine and *b)* the load

generator. The Inference Engine component is responsible for performing the detection and classification tasks. It receives as input the pre-trained DNN model used during inference (e.g. ResNet, Mobilenet, etc.) and the corresponding backend framework (e.g. ONNX Runtime, Tensorflow, etc.). The load generator module is responsible for generating traffic on the inference engine and measuring its performance. It receives as input the validation dataset, (e.g., Imagenet, Coco), as well as the scenario and the number of inference queries to be performed. In our case, the load generator sends 8 samples per query once the previous query is completed. With the above inputs, the load generator performs streaming queries to the inference engine and waits for the results.

Synthetic Interference Generation: To quantify the impact caused by co-locating applications on an inference server, we spawn different amounts of interfering micro-benchmarks, each of which stresses a different resource on the underlying system. Specifically, we utilize the CPU, L2 and L3 cache and memory bandwidth/capacity stress micro-benchmarks, derived from the iBench suite [53]. The duration of each container is set to 1 hour to ensure that the impact of each micro-benchmark on a shared resource increases slowly, to account for the fact that each job generates a small amount of constant intensity interference during our experiments. Consequently, we control the intensity by adjusting the number of concurrent iBench workloads that stress each resource accordingly.

B. MLPerf performance characterization

To gain deeper insights into the execution specifics of the various inference engines listed in Table I, we quantify the impact of different knobs on their performance. Specifically, we profile the various engines, aiming to pinpoint how the *i)* different model representation backends, *ii)* vertical and horizontal resource scaling and *iii)* resource interference affect the performance (queries per second – QPS) of each of the examined engines. Regarding the different backends, we examine the ONNX [54] and Tensorflow [55] model representations, provided directly by MLPerf. Vertical up and down scaling is achieved by altering the environmental variables `OMP_NUM_THREADS` and `INTRA_OP_PARALLELISM_THREADS` for ONNX and Tensorflow backends respectively, which define the number of available thread pools used for parallel executions. These parameters enable operation-level parallelism, e.g., internal parallelization of Tensorflow’s matrix multiplication operation (`tf.matmul()`). Moreover, for horizontal scaling, we further examine the impact of MLPerf’s `#workers` configuration, which controls the number of parallel worker threads, i.e., the amount of worker threads that can serve inference requests simultaneously. Worker threads are instantiated using Python’s `threading` library [56] and run as background daemons that serve requests from a task queue in a first-come first-serve manner.

Isolated Execution: First, we investigate the influence of vertical and horizontal scaling on the performance of different model variants of the investigated ML tasks, i.e., image classification and object detection, without interference. We

TABLE I: Supported MLPerf inference engines, model variant and corresponding accuracy

Task	Representation	Model Variant	Accuracy
Image Classification	ONNX	Resnet50	76.4%
		Mobilenet	71.6%
	Tensorflow (TF)	Resnet50	76.4%
		Mobilenet	71.6%
		Mobinet Quantized(Q)	70.6%
Object Detection	ONNX	SSD-Mobilenet	mAP 0.23
		SSD-Mobilenet	mAP 0.23
	Tensorflow (TF)	SSD-Mobilenet Quantized Fine-Tuned(QFT)	mAP 0.23594
		SSD-Mobilenet Symmetrically Quantized	mAP 0.234
		Fine-Tuned(SQFT)	

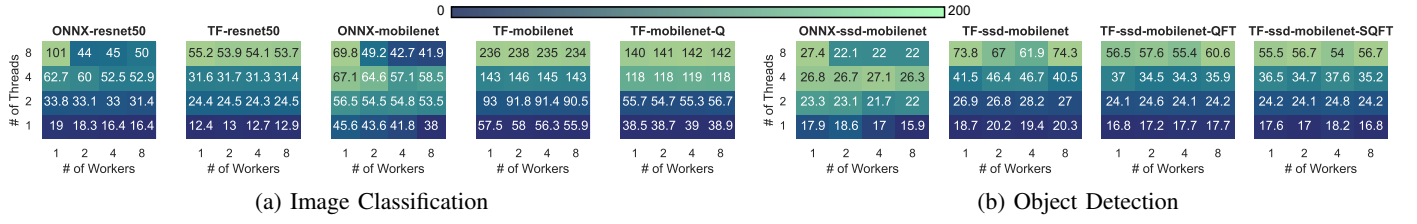


Fig. 1: Impact of #Workers and #Threads on the achieved QPS of the examined inference engines

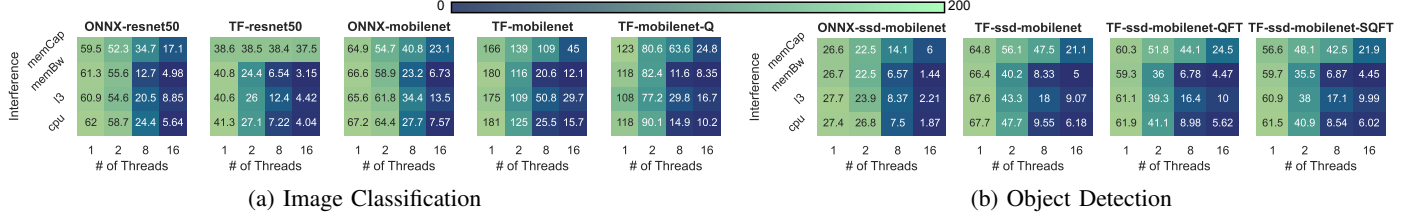


Fig. 2: Impact of different sources and levels of interference on the studied MLPerf inference engines

examine different numbers of workers and threads, ranging from 1 up to 8, where the latter corresponds to fully utilizing our worker VM (8 vCPUs). Figure 1 shows the results of the experiment performed.

★ Q: How do different model variants for the same task behave in terms of performance?

Figure 1 shows that different model variants can achieve different QPS values for the same task. Focusing on single worker/thread performance, we see that, in the image classification task, TF-Mobilenet has the highest QPS, serving $1.26\times$ more queries per second than ONNX-Mobilenet and $4.64\times$ more than TF-Resnet50. Performance differences are also observed in the object detection task, where TF-SSD-Mobilenet has a QPS $1.11\times$ higher than the Quantized Fine-tuned version of the same model. Moreover, while different model representations (i.e., ONNX, TF) do have an impact on performance, there is no clear dominance of one over the other. For example, in the case of resnet50 model (image classification task), ONNX clearly outperforms TF, providing $\approx 1.5\times$ higher throughput, while for mobilenet, TF stands out, with $\approx 1.3\times$ more QPS. Lastly, for the object detection model variants we observe that different model representations do not affect performance dramatically. From the above, it is evident that determining the most suitable DNN architecture and model representation backend is not a trivial process.

★ Q: How does vertical scaling (i.e., #Threads) of resources affect performance?

In terms of the vertical scaling parameters, i.e., the parameter `OMP_NUM_THREADS` for ONNX Runtime and `INTRA_OP_PARALLELISM_THREADS` for Tensorflow (#threads), we find that the inference engine performance increases as we increase the parameter value. Specifically, a parameter value of 8 results in $2.8\times$ higher QPS for the

ONNX Runtime and $3.8\times$ higher QPS for Tensorflow. We also note that the benefits of vertical scaling are different for different model variants and representations. While all the models present high linear correlation between the number of threads and QPS achieved, the final performance improvements are not equally proportional. For example, in the image classification task, a #threads value of 8 for ONNX-Resnet50 results in $5.3\times$ more queries being served per second, as opposed to only $1.5\times$ more queries being served for ONNX-Mobilenet.

★ Q: How does horizontal scaling (i.e., #Workers) of resources affect performance?

The effects of the #workers parameter differ depending on the corresponding backend, i.e. Tensorflow and ONNX Runtime. Figure 1 shows that for TF, increasing the number of workers does not provide any improvement in throughput. This happens due to CPython’s Global Interpreter Lock a.k.a GIL [57]. The GIL ensures that only one thread is running in the interpreter at a time and is designed this way to simplify low-level details such as synchronisation or memory management of concurrent threads. While this leads to high single-threaded performance, it limits parallel or multi-threaded execution of Python code [58]. On the other hand, ONNX is negatively affected when scaling the number of available workers. Specifically, in the case of single thread and multiple workers we observe an average of 14% QPS drop among all examined inference engines. What is of great interest is that, when scaling both the amount of available workers and threads per worker, performance drops dramatically, leading to up to $\times 2$ less throughput. This happens because the total amount of threads from all workers ($\#workers \times \#threads$) exceed the number of available CPU resources of the system, leading to continuous context switching in the OS, and thus generating resource interference on the CPU.

Execution under Interference: Next, we further examine the performance of the different inference engines under the presence of interference. To illustrate the effects of resource contention, we perform a series of experiments in which we measure the QPS that each model achieves while the server is stressed at a particular shared resource at a time, as well as the variance of the QPS value when these resources are stressed at different intensities (1 to 16 iBench containers).

★ Q: How does resource interference affect the performance of the inference engines?

Figure 2 shows the effects of various sources of interference on the studied MLPerf inference engines. We observe that, for the general case, the higher the number of parallel iBench jobs, the greater the QPS degradation. However, performance degradation due to the increasing number of co-located micro-benchmarks is not uniform across all the levels of resource interference. Specifically, for the inference engines examined, memory bandwidth and CPU stress have the greatest influence on the QPS achieved, while last-level cache and memory capacity follow. In the image classification task, using 16 parallel iBench jobs results in an average $10.65\times$ lower QPS compared to using one iBench job when the CPU and $6.66\times$ when stressing the last-level cache. For the object detection task, inducing the same interference results in an average $11.71\times$ lower QPS when stressing the CPU, and $8\times$ when stressing the last-level cache.

An interesting observation regarding the TF-Resnet50 inference engine is that memory capacity stress has no impact on throughput, showing that the impact of interference also depends on application characteristics.

★ Q: Do different backends (i.e., ONNX, TF) reveal different performance sensitivity w.r.t. resource interference?

Figure 2 also shows the impact of the backend used, i.e., ONNX Runtime and Tensorflow, on the QPS achieved. It can be seen that the Resnet50 implementation in the ONNX Runtime environment is able to achieve higher QPS than the Tensorflow implementation for the same level of interference. For example, when the CPU is stressed from one iBench job, ONNX-Resnet50 performs $1.5\times$ more queries per second compared to TF-Resnet50. The picture changes for the corresponding implementations of Mobilenet and SSD-Mobilenet, where the Tensorflow implementations outperform the ONNX Runtime in terms of QPS. In particular, for the interference scenario mentioned above, the TF-Mobilenet and TF-SSD-Mobilenet achieve $2.7\times$ and $2.5\times$ higher QPS, respectively, compared to the corresponding ONNX Runtime model variants.

★ Q: How do different resource allocations affect the performance of the inference engines under the presence of interference?

Aiming to highlight the effects of vertical and horizontal scaling in the presence of interference, we also perform

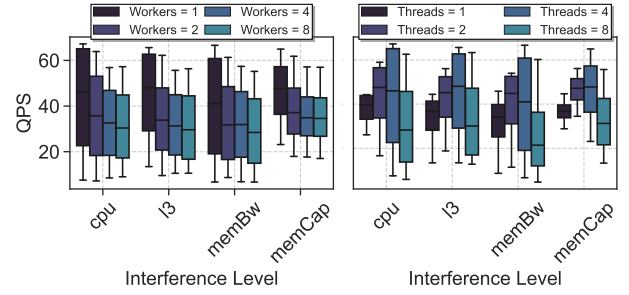


Fig. 3: Impact of #OMP and #MLPerf threads on ONNX Mobilenet under interference

experiments in which we change the tuning parameters. For the ONNX backend, we maintain one of the two tuning parameters constant, and change the value of the other, to see how the freely varying variable affects the QPS value and its variance under interference. Thus, we conduct experiments by varying only one of the variables, respectively. For the Tensorflow backend, we only consider the tuning parameter `#threads`, since the number of workers did not provide any performance variability in the isolated characterization process.

Figure 3 shows the results for the ONNX backend. Focusing on the left subplot, where the environment variable `OMP_NUM_THREADS` is held constant, we see that we get higher QPS when the `#workers` parameter is equal to 1, indicated by the darkest box (`#workers = 1`), than when we use a higher value. In the subplot where the `#workers` option is held constant, the best performance is observed for `#threads` values of 2 and 4, while a QPS drop appears to occur for a `#threads` value of 8. In addition, Figure 4 illustrates the results for the corresponding tensorflow backend. Initially, we observe a similar upward trend in performance, for all MLPerf Inference engines with the Tensorflow backend, as the value of `#threads` increases. The highest QPS values are generally found when the number of threads exceeds 5, where the median QPS value goes up to 90. The lowest performance variability is observed when the Tensorflow threads are equal to 1, for all different sources of interference. Moreover, as the stress intensity in memory capacity varies, the performance of the models remains quite robust and deviates only slightly from the median value of the achieved QPS, compared to the high performance variability observed with the varying degrees of pressure on the other shared resources.

IV. IRIS DESIGN

Based on the insights gained from the characterization process, we design IRIS. Its main goal is to meet the QoS requirements of the deployed inference engines while minimizing the allocated CPU resources of the system. IRIS identifies interference effects by exploiting low-level performance events, and leverages these metrics to provide accurate predictions regarding the performance of the deployed inference engines in the future. Based on these predictions, it automatically assigns resources and dynamically applies horizontal and vertical scaling policies to the deployed inference engines. Our design

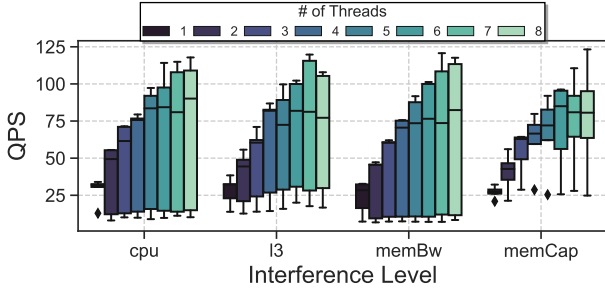


Fig. 4: Impact of #Tensorflow threads on Tensorflow Mobilenet Quantized under interference

approach consists of two distinct phases: a) the offline phase and b) the online phase, as shown in Figure 5, which we explain in detail in the following sections.

A. Offline Phase

The purpose of the offline phase is twofold. The first stage is the training data preparation life-cycle for building the ML models used by IRIS. This includes the generation of different co-location scenarios, data extraction, cleanup, aggregation and feature engineering. Overall, training data should include representative scenarios that also exist under realistic deployments and cover a sufficient spectrum of application behaviors, so as to avoid covariate shift between training and run-time (testing) data distributions [59]. The second phase includes the identification of the most efficient ML approach for modeling the performance, in terms of QPS prediction accuracy, per inference engine. Deciding the most appropriate ML algorithm is not a straightforward matter, as it highly depends on the quality of the gathered data, the inherent characteristics and complexity of the problem, as well as the level of accuracy and interpretability required [60]. Thus, IRIS performs a design space exploration (DSE) over a pool of different ML solutions to identify the most appropriate ML modeling approach, as well as the hyperparameters that are used during the learning process.

Scenario Generation: This step concerns the generation of representative co-location scenarios, which cover both low- and high-intensity interference across different layers of the system. We use asymmetric interference load generation on the server, by co-locating MLPerf inference engines along with iBench [53] interference micro-benchmarks. Specifically, each scenario consists of initially deploying a random number of iBench containers, each of which occupies a randomly selected specific shared resource of the cluster, i.e., CPU, L2 cache, L3 cache, memory bandwidth, and memory capacity, for the duration of the scenario. The total number of iBench containers used varies between 1 and 16. After setting the interference conditions, we randomly select one of the available inference engines, either for image classification or for object detection, and deploy it on the worker server. Moreover, we randomly set the horizontal and vertical degree of parallelism, whose values range from 1 up to 8. As mentioned in section III, each inference engine uses the Multiple Stream scenario,

with duration limited to 60 seconds. These parameters fully describe the execution scenario 1a. In total, we have run more than 5500 different deployment scenarios, with each scenario differing by the above parameters.

Performance Monitoring: The monitoring mechanisms of IRIS are driven by two main principles. First, the resource orchestration component should be able to assess the state of the underlying system, in terms of existing interference, prior and during the execution of the inference engines. As shown in Section III, underlying interference highly affects the performance of inference serving and this “interference state” can be used as input to the ML prediction models. To capture interference effects, we exploit low-level system performance events (e.g., IPC, Cache misses, etc.) as well as higher-level metrics (e.g., CPU/Memory utilization and others) 1b, which, as shown in prior research, provide deep insights regarding resource contention and highly correlate with the performance of applications [25], [61]. Second, IRIS should be able to forecast the future QPS of the inference engines under dynamically altered interference effects. These predictions can be used to pro-actively alter allocated resources per inference engine, minimizing QoS violations. Thus, IRIS also continuously monitors application-related performance metrics (i.e., QPS 1c). While MLPerf provides application-related performance metrics advertisement, we assume that, in general, this operation is handled by application developers, who should build and propagate the respective APIs to be consumed by IRIS. We set the monitoring interval to 1 second, both for the system and the inference engine monitoring components. Although more frequent monitoring intervals could provide faster detection of QoS violations, they can also introduce noisy and unstable results, leading to false alarms regarding potential violations [20].

Training Dataset Formation: For building the training dataset, IRIS collects the different system metrics for each MLPerf inference engine under different interference scenarios. The random scenario execution data is accumulated to form the training dataset 2a. Each row of the dataset contains i) the name of the selected inference engine, ii) the backend framework, i.e., Tensorflow or ONNX, iii) the system metrics collected at the beginning of each training scenario, iv) the assigned values of the `INTRA_OP_PARALLELISM_THREADS` for the Tensorflow backend and the `OMP_NUM_THREADS` and `--threads` options for the ONNX backend (`#threads`), and v) the QPS achieved.

Performance Modeling DSE: The purpose of this step is to identify the most efficient ML model, per inference engine, for predicting the QPS achieved under interference. From the training dataset 2b, we isolate each time the rows pertaining to a single engine and train a number of different regression models 2c, each with its default training hyperparameters 2d. Finally, we compare the accuracy of their prediction of QPS using the result of a 10-fold cross-validation of the scikit-learn Python library [62] to identify the best model 2e. Table IV shows the most striking results for the achieved accuracy when the trained dataset for each engine is given as input to

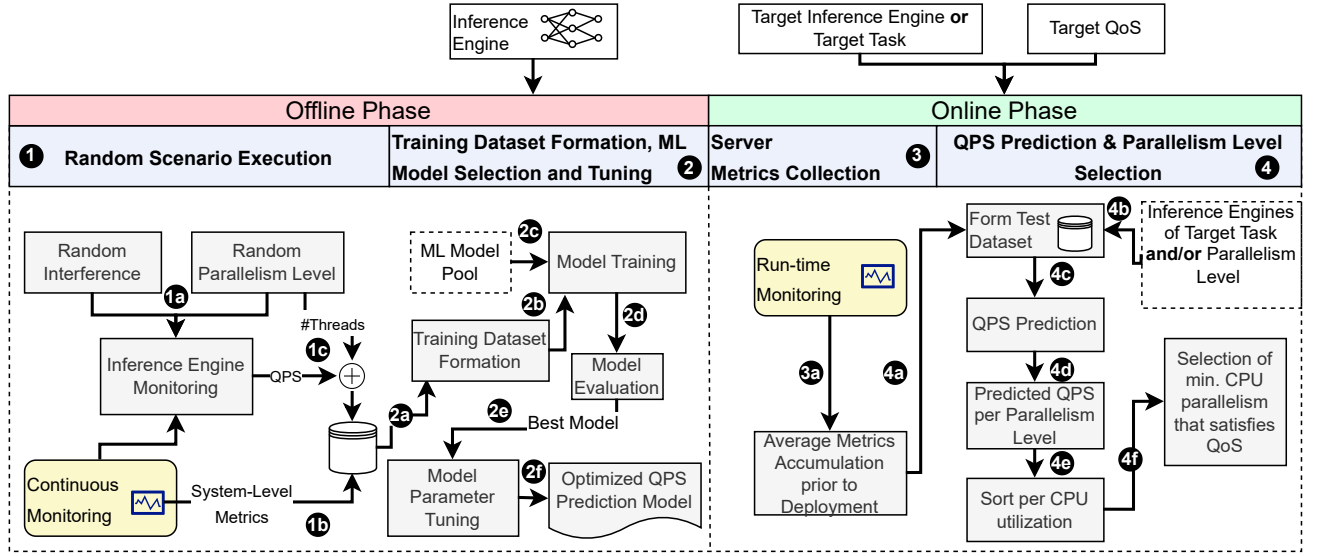


Fig. 5: Overview of the offline and online scheduling phase of IRIS

TABLE II: XGBoost regression best parameters after hyper-parameter tuning for each MLPerf inference engine

	Model Variant	Subsample	Silent	Reg_lambda	Estimators	Min_Child_Weight	Max_Depth	Learning_Rate	Gamma	Colsample_bytree	Colsample_bylevel
Onnx	Resnet50	0.7	False	100.0	1600.0	1.0	10.0	0.2	0.5	0.8	1.0
	Mobilenet	0.7	False	5.0	1600.0	0.5	6.0	0.01	1.0	0.6	0.9
	Ssd-Mobilenet	0.5	False	5.0	800.0	10.0	20.0	0.1	0.5	0.8	0.8
	Resnet50	0.8	False	50.0	400.0	5.0	15.0	0.1	0.5	1.0	0.7
Tensorflow	Mobilenet	0.5	False	50.0	3200.0	1.0	20.0	0.1	0.5	1.0	0.9
	Mobilenet-Q	0.8	False	50.0	3200.0	5.0	15.0	0.01	0.5	0.9	0.8
	Ssd-Mobilenet	1.0	False	50.0	3200.0	5.0	6.0	0.1	0.25	1.0	0.9
	Ssd-Mobilenet-QFT	0.8	False	5.0	100.0	3.0	15.0	0.1	0.0	1.0	0.7
	Ssd-Mobilenet-SQFT	0.9	False	10.0	3200.0	1.0	10.0	0.01	0.5	1.0	0.5

models in the ML model pool. As we can see from the tables, the most accurate ML regression models in predicting the performance of the MLPerf inference benchmarks are Random Forest Regression and XGBoost Regression. This is true for both the image classification task and the object detection task. It can also be seen that the XGBoost regression in particular has the best overall score for almost all inference engines. Since the Random Forest regression outperforms the XGBoost regression in only 2 of the total 9 predictions, and that only slightly, we will use the XGBoost regressor for predicting QPS for all MLPerf inference benchmarks.

Model's hyperparameter tuning: To further increase the score by cross-validation obtained with the XGBoost regressor in the default configuration, we perform a hyperparameter optimization process (2f) to select a set of hyperparameters for our regressor, tailored to the performance of each MLPerf inference benchmark it needs to predict. We perform a randomized search for the hyperparameters of the XGboost regressor, for each MLPerf Inference benchmark, using the RandomizedSearchCV function of the scikit-learn Python library. Table II contains the best parameters for the XGBoost ML Regression algorithm, for each MLPerf inference engine, after the hyperparameter tuning while Table III summarizes the default and tuned accuracy achieved. Adjusting the model parameters leads to an average increase in accuracy of 1.5%.

B. Online Phase

In the online phase IRIS continuously evaluates the levels of interference on the underlying system, by monitoring the hardware-related monitors described in Section IV-A. Based on these monitors, it performs QPS predictions for newly deployed inference engines, as well as running ones, and dynamically re-configures their allocated resources, in order to minimize CPU utilization while satisfying a user-defined target QPS value. IRIS supports two different inference serving approaches: i) the model-specific one, where the specific inference engine that the end-user wants to use, serves as input along with the required QoS, and ii) the model-less approach, where only the task, i.e., image classification or object detection, serves as input and it is decided which model is best to use to satisfy the required QoS.

Model-specific: In the online phase of the model-specific approach, as mentioned earlier, our scheduler takes the target inference engine with the target QoS as input and attempts to determine the degree of parallelism that will allow the system to be as resource-constrained as possible, while still satisfying the required QoS constraint. Through continuous monitoring, our mechanism gains insight into the current level of disruption using various system metrics such as CPU and cluster memory usage, which are collected over a predefined interval (3a), and the average values per metric are calculated.

Depending on which backend, i.e. Tensorflow or ONNX, the

inference engine uses, it sets the values for the corresponding inference engine tuning knobs. To achieve this, our scheduler relies on the pre-trained XGBoost model. Specifically, given the interference conditions indicated by the collected system metrics (4a) and all the different parallelization levels (4b), the regression model predicts the QPS (4c) that the benchmark would achieve. After collecting the predicted QPS scores for the different parallelization levels (4d), we perform a sort, based on the CPU utilization (4e), and select the tuning knob value that minimizes the CPU utilization and satisfies the QoS (4f).

Model-less: In the online phase of the modelless approach, the end user specifies only the task he wants to perform, i.e., image classification or object detection, and the target QoS. The modelless scheduler then selects the best inference engine with the appropriate degree of parallelism from a task-specific pool of registered, trained inference models to complete the task in the least resource-intensive manner, while satisfying the QoS constraint under the current interference conditions in the system. The main difference is that during the QPS prediction phase, the model-less scheduler not only tries different levels of parallelism, but also different models for the given task. Apart from this additional degree of freedom, the other steps are identical.

V. EVALUATION

To evaluate our schedulers, we perform a series of tests on both the model-specific and model-less approaches. Each scheduling system is evaluated against different baselines, different levels of interference, and different QoS requirements.

A. Model-specific Inference Engine Scheduler Evaluation

Experiment Description: To quantify the impact of different levels of interference on scheduling decisions, we create 3 different interference scenarios. In each scenario, iBench batches are deployed on our cluster, pressuring various shared resources for a specified time interval ranging from 70 to 220 seconds. When an iBench batch is finished, the next one is started after a random time interval between 10 and 30 seconds. After the interference level is determined, the inference engines presented in section III are used to provide a specific QoS constraint. Each engine is deployed 10 times

TABLE III: Score of 10-fold cross-validation comparison of the XGBoost Regression before and after hyper-parameter optimization

	Model Variant	Default	Tuned
Onnx	Resnet50	0.843	0.877 (+3.8%)
	Mobilenet	0.914	0.924 (+1.08%)
	Ssd-Mobilenet	0.941	0.943 (+0.21%)
Tensorflow	Resnet50	0.962	0.964 (+0.20%)
	Mobilenet	0.962	0.97 (+0.82%)
	Mobilenet-Q	0.956	0.966 (+1.03%)
	Ssd-Mobilenet	0.965	0.971 (+0.61%)
	Ssd-Mobilenet-QFT	0.956	0.964 (+0.82%)
	Ssd-Mobilenet-SQFT	0.961	0.966 (+0.51%)

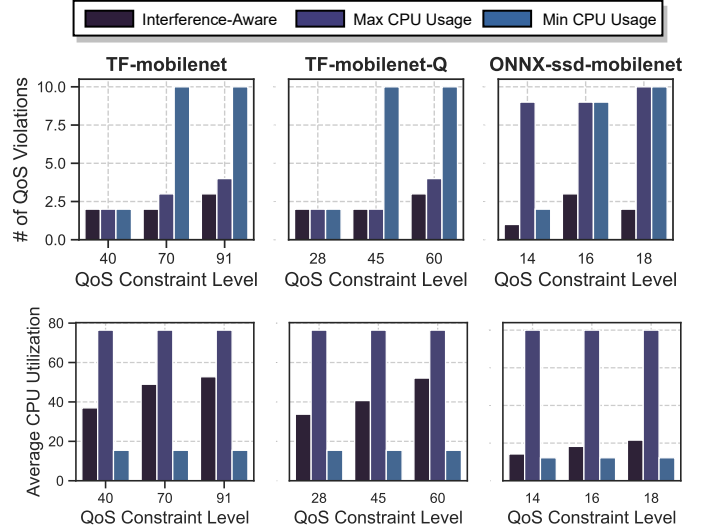


Fig. 6: Model-specific inference serving evaluation of the number of QoS violations(top) and average CPU utilization(bottom) over different QoS constraint levels.

for a duration of 30 seconds. We also consider 3 different QoS constraints i) Low, ii) Medium, and iii) High for each inference engine, based on the QPS values it can achieve. Finally, we compare our model-specific scheduler to two different schedulers: i) a scheduler that provides a minimum of CPU resources (Min CPU Usage), and ii) a scheduler that provides full-provisioning and thus maximizes the utilization of CPU (Max CPU Usage).

Figure 6 (top) shows the violations of the specified QoS constraint for 3 representative inference engines i) the TF-Mobilenet, ii) the TF-Mobilenet-Q, and iii) the ONNX-SSD-Mobilenet. X axis shows different QoS constraint levels, while Y axis depicts the number of QoS violations. Our interference-aware scheduler manages to keep the number of QoS violations significantly lower than Min CPU Usage in all cases with medium or high QoS constraints, and performs better or equal to Min CPU Usage in almost all cases with low QoS constraints. IRIS (Interference-Aware) achieves 67.53% less QoS violations compared to Min CPU Usage, on average. Furthermore, our scheduler manages to perform remarkably better than the Max CPU Usage in the ONNX SSD-Mobilenet

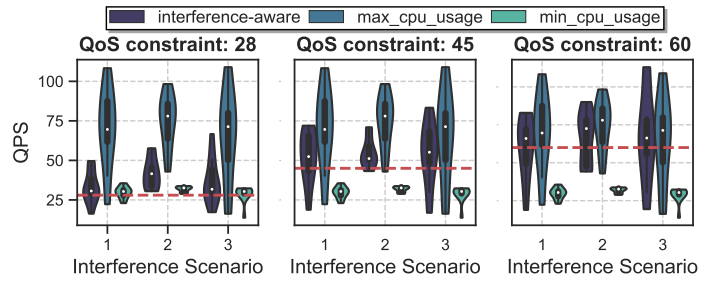


Fig. 7: QPS Distributions for TensorFlow Mobilenet Quantized over different QoS constraints and interference scenarios

TABLE IV: Evaluation by 10-fold cross-validation of different regression models for difference ML engines

Model	Image Classification					Object Detection			
	ONNX		Tensorflow			ONNX	Tensorflow		
	Resnet50	Mobilenet	Resnet50	Mobilenet	Mobilenet-Q	Ssd-mobilenet	Ssd-mobilenet	Ssd-mobilenet-QFT	Ssd-mobilenet-SQFT
Lasso	0.679	0.801	0.745	0.783	0.797	0.838	0.792	0.763	0.769
Elastic Net	0.679	0.803	0.744	0.783	0.798	0.842	0.791	0.762	0.765
Decision Tree	0.632	0.835	0.921	0.93	0.92	0.878	0.927	0.926	0.929
RF	0.825	0.909	0.955	0.962	0.959	0.942	0.963	0.954	0.961
XGBoost	0.843	0.914	0.962	0.962	0.956	0.941	0.965	0.956	0.961

inference engine. More specifically, we achieve, on average, 43.97% less QoS violations compared to the *Max CPU Usage* approach. This is because the ONNX benchmarks perform best at moderate parallelism, while they show a performance drop at high parallelism. In the Tensorflow benchmarks, the number of QoS violations of our custom scheduler is similar to that of the *Max CPU Usage*, although our scheduler achieves this without using the cluster's resources at maximum capacity all the time.

Figure 6 (bottom) depicts the average CPU utilization of the studied scheduling approaches for different QoS constraint levels. Our interference-aware scheduler achieves 36.78% CPU utilization, on average which is between the CPU utilization of the *Min CPU Usage* (15.38%) and the *Max CPU Usage* (all threads occupied - 84.27%). This means that our custom scheduler is able to meet or exceed the QoS constraint for most inference engines by utilizing 56.4% less CPU resources compared to *Max CPU Usage*, and avoiding the resource under-utilization and high QoS violations, compared to *Min CPU Usage*.

To gain further insight into the behaviour of the examined scheduling approaches, we examine the QPS distributions obtained for i) the TF-Mobilenet-Q and ii) the ONNX-SSD-Mobilenet Inference Engines, respectively, for 3 different QoS targets and 3 different interference scenarios. As Figure 7 shows, our interference-sensitive scheduler exhibits 26% less QPS compared to *Max CPU Usage* for TF-Mobilenet-Q. However, this comes with the cost of resource over-utilization. For ONNX-SSD-Mobilenet in figure 8, our scheduler achieves 44% more QPS, on average, thus we confirm that maximizing resource utilization does not necessarily lead to high QPS due to context switching and congestion. Moreover, we see a performance distribution with the highest probability of QPS

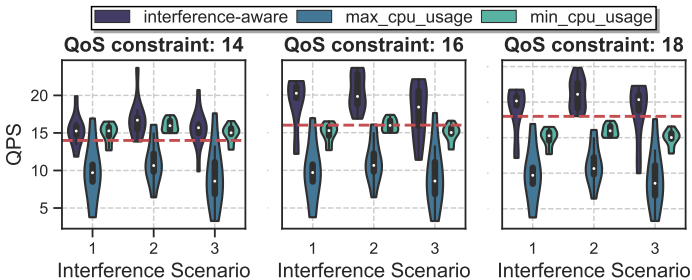


Fig. 8: QPS Distributions for ONNX Runtime SSD-Mobilenet over different QoS constraints and interference scenarios

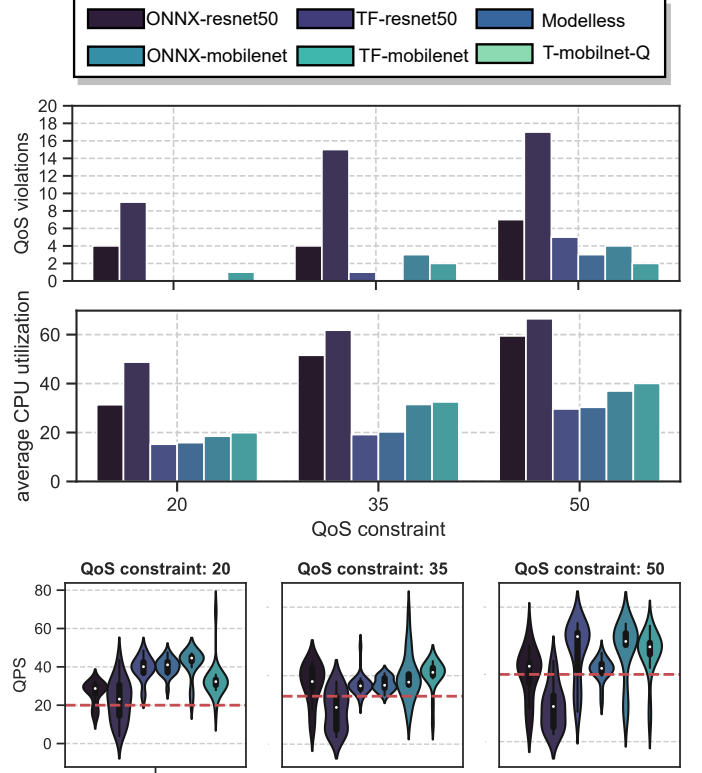


Fig. 9: Model-less Inference Serving Evaluation for QoS violations(top), average CPU utilization(middle) and QPS(bottom) for Image Classification Task.

values above the target QoS for all QoS constraints. Therefore, our scheduler provides more QoS guarantees compared to *Min CPU Usage*, by achieving 59.2% and 16.6% higher QPS, on average, for TF-Mobilenet-Q and ONNX-SSD-Mobilenet, respectively. Thus, IRIS can guarantee high QPS, while avoiding resource congestion.

B. Model-less Inference Engine Scheduler Evaluation

Experiment Description: When evaluating the model-less scheduler, we only specify the desired task, i.e., image classification or object detection, and the scheduler selects the appropriate inference engine and the values of the tuning knobs. We consider 20 requests, the duration of each inference engine being 30 seconds. Finally, our model-less scheduler is compared to all different interference-aware model-specific schedulers for the given task e.g., for the object detection task,

a baseline is the interference-aware model-specific scheduler using only the TF-SSD-Mobilenet inference engine.

Figure 9 indicates the QoS violations(top) and average CPU utilization(middle) for the image classification task. For low(20) and medium(35) target QoS constraint, the model-less scheduler mostly uses the ONNX-Mobilenet benchmark, which never violates the constraint. With the high QoS constraint(50), most of the time the TF-Mobilenet benchmark is used, which has a higher QPS capability with a lower CPU utilization. As a result, the model-less scheduler achieves, on average, 57.63% less QoS violations compared to the others, where a single model variant is utilized. Furthermore, as we observe in Figure 9(middle) the model-less approach achieves low QoS violations, while keeping the average CPU utilization down to 21.34%, on average, across all QoS constraints. Moreover, as it can be seen from the QPS distributions in Figure 9, the model-less scheduler exhibits similar performance variability to the benchmark it uses most often for each QoS objective. More specifically, it manages to keep the mean QPS value above the QoS constraints, while it achieves $2\times$, $1.22\times$ and $1.56\times$ higher QPS for low, medium and high QoS constraint, on average.

In a similar experiment for object detection tasks, Figure 10 highlights the QoS violations(top), the average CPU utilization(middle) and the QPS(bottom), respectively. In particular, our model-less scheduler mainly uses the ONNX-SSD-Mobilenet benchmark with a low QoS constraint, by achieving up to 23.89% less QoS violations, on average, for all the QoS constraints (Fig. 10 top). For the medium and high QoS requirements, the model-less scheduler applies the TF-SSD-Mobilenet benchmark for most deployments, so that, on average, 8 QoS violations occur, as with the model-specific scheduler serving TF-SSD-Mobilenet. However, it performs significantly better than the scheduler with the ONNX-SSD-Mobilenet benchmark in these QoS conditions, by achieving 42.36% less QoS violations, on average. Furthermore, the average CPU utilization is maintained at a minimum for all QoS conditions, by consuming down to 34.19% of the total amount of the CPU (Fig. 10 middle). The performance variability shown in Figure 10 is similar to the performance variability it mostly selects, resulting in a mean QPS score above the QoS constraints (Fig. 10 bottom).

VI. CONCLUSION

In this work, we present an interference- and resource-aware predictive scheduling framework for ML inference engines that aims to guarantee application-specific QoS constraints, while minimizing resource utilization. Our framework is based on a model-less approach and is integrated with the Kubernetes system. Under various QoS constraints, the model-specific interference-aware scheduler violates QoS constraints less frequently by achieving $1.8\times$ fewer violations, on average, compared to over-provisioning and $3.1\times$ fewer violations compared to under-provisioning, through efficient exploitation of available CPU resources. The model-less feature is able to cause, on average, $1.5\times$ fewer violations compared to the

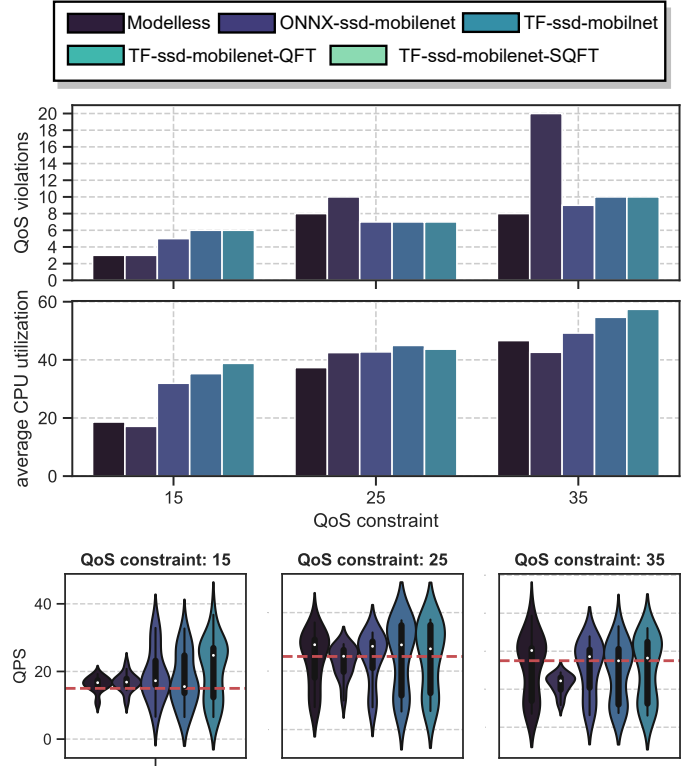


Fig. 10: Model-less Inference Serving Evaluation for QoS violations(top), average CPU utilization(middle) and QPS(bottom) for Object Detection Task.

model-specific scheduler, while further reducing the average CPU utilization by $\approx 30\%$.

REFERENCES

- [1] S. Pouyanfar, S. Sadiq, Y. Yan, H. Tian, Y. Tao, M. P. Reyes, M.-L. Shyu, S.-C. Chen, and S. S. Iyengar, "A survey on deep learning: Algorithms, techniques, and applications," *ACM Computing Surveys (CSUR)*, vol. 51, no. 5, pp. 1–36, 2018.
- [2] C. Shorten, T. M. Khoshgoftaar, and B. Furht, "Deep learning applications for covid-19," *Journal of big Data*, vol. 8, no. 1, pp. 1–54, 2021.
- [3] A. Arévalo, J. Niño, G. Hernández, and J. Sandoval, "High-frequency trading strategy based on deep neural networks," in *Intelligent Computing Methodologies: 12th International Conference, ICIC 2016, Lanzhou, China, August 2-5, 2016, Proceedings, Part III 12*, pp. 424–436, Springer, 2016.
- [4] X. Ma, H. Yu, Y. Wang, and Y. Wang, "Large-scale transportation network congestion evolution prediction using deep learning theory," *PLoS one*, vol. 10, no. 3, p. e0119044, 2015.
- [5] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.
- [6] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al., "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.
- [7] "Deliver high performance ML inference with AWS Inferentia." https://d1.awsstatic.com/events/reinvent/2019/REPEAT_1_Deliver_high_performance ML_inference_with_AWS_Inferentia_CMP324-R1.pdf. Accessed: 04-03-2023.
- [8] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, et al., "Applied machine learning at facebook: A datacenter infrastructure perspective," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 620–629, IEEE, 2018.

- [9] "IBM Watson." <https://www.ibm.com/watson>, note = Accessed: 04-03-2023.
- [10] "Vertex AI." <https://cloud.google.com/vertex-ai>. Accessed: 04-03-2023.
- [11] "Amazon SageMaker." <https://aws.amazon.com/sagemaker/>. Accessed: 04-03-2023.
- [12] "Azure Machine Learning." <https://azure.microsoft.com/en-us/products/machine-learning>. Accessed: 04-03-2023.
- [13] A. Gujarati, S. Elnikety, Y. He, K. S. McKinley, and B. B. Brandenburg, "Swayam: distributed autoscaling to meet slas of machine learning inference services with resource efficiency," in *Proceedings of the 18th ACM/IFIP/USENIX middleware conference*, pp. 109–120, 2017.
- [14] C. Zhang, M. Yu, W. Wang, and F. Yan, "Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving," in *USENIX Annual Technical Conference*, pp. 1049–1062, 2019.
- [15] J. Guo, Z. Chang, S. Wang, H. Ding, Y. Feng, L. Mao, and Y. Bao, "Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces," in *Proceedings of the International Symposium on Quality of Service*, pp. 1–10, 2019.
- [16] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.
- [17] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 77–88, 2013.
- [18] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pp. 450–462, 2015.
- [19] J. Leverich and C. Kozyrakis, "Reconciling high server utilization and sub-millisecond quality-of-service," in *Proceedings of the Ninth European Conference on Computer Systems*, pp. 1–14, 2014.
- [20] S. Chen, C. Delimitrou, and J. F. Martínez, "Parties: Qos-aware resource partitioning for multiple interactive services," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 107–120, 2019.
- [21] D. Crankshaw, G.-E. Sela, X. Mo, C. Zumar, I. Stoica, J. Gonzalez, and A. Tumanov, "Inferline: latency-aware provisioning and scaling for prediction serving pipelines," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, pp. 477–491, 2020.
- [22] L. Suresh, M. Canini, S. Schmid, and A. Feldmann, "C3: Cutting tail latency in cloud data stores via adaptive replica selection," in *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pp. 513–527, 2015.
- [23] X. Ma, J. Li, M. J. Kochenderfer, D. Isele, and K. Fujimura, "Reinforcement learning for autonomous driving with latent state inference and spatial-temporal relationships," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 6064–6071, IEEE, 2021.
- [24] Y. Liu, L. Jiao, Y. Liu, and J. He, "A self-adapting fuzzy inference system for the evaluation of agricultural land," *Environmental modelling & software*, vol. 40, pp. 226–234, 2013.
- [25] D. Masouros, S. Xydis, and D. Soudris, "Rusty: Runtime interference-aware predictive monitoring for modern multi-tenant systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 1, pp. 184–198, 2020.
- [26] A. Tzenetopoulos, D. Masouros, S. Xydis, and D. Soudris, "Interference-aware orchestration in kubernetes," in *High Performance Computing: ISC High Performance 2020 International Workshops, Frankfurt, Germany, June 21–25, 2020, Revised Selected Papers 35*, pp. 321–330, Springer, 2020.
- [27] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis, "Infraas: Automated model-less inference serving," in *USENIX Annual Technical Conference*, pp. 397–411, 2021.
- [28] "Tensorflow Serving." <https://www.tensorflow.org/tfx/guide/serving>. Accessed: 01-03-2023.
- [29] "Triton Inference Server." <https://developer.nvidia.com/nvidia-triton-inference-server>. Accessed: 01-03-2023.
- [30] A. Ali, R. Pincirollo, F. Yan, and E. Smirni, "Batch: Machine learning inference serving on serverless platforms with adaptive batching," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15, IEEE, 2020.
- [31] B. Fu, F. Chen, P. Li, and D. Zeng, "Tcb: Accelerating transformer inference services with request concatenation," in *Proceedings of the 51st International Conference on Parallel Processing*, pp. 1–11, 2022.
- [32] Y. Yang, L. Zhao, Y. Li, H. Zhang, J. Li, M. Zhao, X. Chen, and K. Li, "Infless: a native serverless system for low-latency, high-throughput inference," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 768–781, 2022.
- [33] N. Mahmoudi and H. Khazaei, "Mlproxy: Sla-aware reverse proxy for machine learning inference serving on serverless computing platforms," *arXiv preprint arXiv:2202.11243*, 2022.
- [34] J. R. Gunasekaran, C. S. Mishra, P. Thinakaran, B. Sharma, M. T. Kandemir, and C. R. Das, "Cocktail: A multidimensional optimization for model serving in cloud," in *USENIX NSDI*, pp. 1041–1057, 2022.
- [35] K. Razavi, M. Luthra, B. Koldehofe, M. Mühlhäuser, and L. Wang, "Fa2: Fast, accurate autoscaling for serving deep learning inference with sla guarantees," in *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 146–159, IEEE, 2022.
- [36] X. Tang, P. Wang, Q. Liu, W. Wang, and J. Han, "Nanily: A qos-aware scheduling for dnn inference workload in clouds," in *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pp. 2395–2402, IEEE, 2019.
- [37] M. Yu, Z. Jiang, H. C. Ng, W. Wang, R. Chen, and B. Li, "Gillis: Serving large neural networks in serverless functions with automatic model partitioning," in *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pp. 138–148, IEEE, 2021.
- [38] C. Zhang, M. Yu, W. Wang, and F. Yan, "Enabling cost-effective, slo-aware machine learning inference serving on public cloud," *IEEE Transactions on Cloud Computing*, vol. 10, no. 3, pp. 1765–1779, 2020.
- [39] Z. Wang, X. Tang, Q. Liu, and J. Han, "Jily: Cost-aware autoscaling of heterogeneous gpu for dnn inference in public cloud," in *2019 IEEE 38th International Performance Computing and Communications Conference (IPCCC)*, pp. 1–8, IEEE, 2019.
- [40] D. Mendoza, F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis, "Interference-aware scheduling for inference serving," in *Proceedings of the 1st Workshop on Machine Learning and Systems*, pp. 80–88, 2021.
- [41] A. Bhattacharjee, A. D. Chhokra, Z. Kang, H. Sun, A. Gokhale, and G. Karsai, "Barista: Efficient and scalable serverless serving system for deep learning prediction services," in *2019 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 23–33, IEEE, 2019.
- [42] L. Wang, L. Yang, Y. Yu, W. Wang, B. Li, X. Sun, J. He, and L. Zhang, "Morphling: Fast, near-optimal auto-configuration for cloud-native model serving," in *Proceedings of the ACM Symposium on Cloud Computing*, pp. 639–653, 2021.
- [43] H. Qin, S. Zawad, Y. Zhou, L. Yang, D. Zhao, and F. Yan, "Swift machine learning model serving scheduling: a region based reinforcement learning approach," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–23, 2019.
- [44] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A low-latency online prediction serving system," in *NSDI*, vol. 17, pp. 613–627, 2017.
- [45] N. J. Yadwadkar, F. Romero, Q. Li, and C. Kozyrakis, "A case for managed and model-less inference serving," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, pp. 184–191, 2019.
- [46] J. Zhang and R. J. Figueiredo, "Application classification through monitoring and learning of resource consumption patterns," in *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, pp. 10–pp, IEEE, 2006.
- [47] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," *ACM Sigplan Notices*, vol. 45, no. 3, pp. 129–142, 2010.
- [48] J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa, "Contention aware execution: online contention detection and response," in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pp. 257–265, ACM, 2010.
- [49] T. Patel and D. Tiwari, "Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 193–206, IEEE, 2020.
- [50] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and qos-aware cluster management," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 127–144, 2014.
- [51] A. Tzenetopoulos, D. Masouros, S. Xydis, and D. Soudris, "Interference-aware workload placement for improving latency distribution of converged hpc/big data cloud infrastructures," in *Embedded Computer Systems: Architectures, Modeling, and Simulation: 21st International*

Conference, SAMOS 2021, Virtual Event, July 4–8, 2021, *Proceedings*, pp. 108–123, Springer, 2022.

- [52] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou, “MLPerf Inference Benchmark,” 2019.
- [53] C. Delimitrou and C. Kozyrakis, “ibench: Quantifying interference for datacenter applications,” in *2013 IEEE international symposium on workload characterization (IISWC)*, pp. 23–33, IEEE, 2013.
- [54] “ONNX Runtime.” <https://onnxruntime.ai/>. Accessed: 24-03-2023.
- [55] “Tensorflow.” <https://www.tensorflow.org/>. Accessed: 24-03-2023.
- [56] “Python Threading Library.” <https://docs.python.org/3/library/threading.html>. Accessed: 24-03-2023.
- [57] D. Beazley, “Understanding the python gil,” in *PyCON Python Conference. Atlanta, Georgia*, 2010.
- [58] R. Meier and A. Rigo, “A way forward in parallelising dynamic languages,” in *Proceedings of the 9th International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems PLE*, pp. 1–4, 2014.
- [59] Y. Zhang, W. Hua, Z. Zhou, G. E. Suh, and C. Delimitrou, “Sinan: ML-based and qos-aware resource management for cloud microservices,” in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS ’21*, (New York, NY, USA), p. 167–181, Association for Computing Machinery, 2021.
- [60] P. Harrington, *Machine learning in action*. Simon and Schuster, 2012.
- [61] R. Brondolin and M. D. Santambrogio, “A black-box monitoring approach to measure microservices runtime performance,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 17, no. 4, pp. 1–26, 2020.
- [62] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, “Scikit-learn: Machine learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.