Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων
και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Interference and Resource Aware Predictive Inference Serving on Cloud Infrastructures

Παναγιώτης Δ. Χρυσομέρης

Επιβλέπων: Δημήτριος Ι. Σούντρης
Καθηγητής ΕΜΠ

Αθήνα
Νοέμβριος 2022

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Interference and Resource Aware Predictive Inference Serving on Cloud Infrastructures

Παναγιώτης Δ. Χρυσομέρης

Επιβλέπων: Δημήτριος Ι. Σούντρης
Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 15η Νοεμβρίου 2022.

(Υπογραφή)	(Υπογραφή)	(Υπογραφή)
…………………………	…………………………	…………………………
Δημήτριος Σούντρης	Παναγιώτης Τσανάκας	Σωτήριος Ξύδης
Καθηγητής	Καθηγητής	Επίκουρος Καθηγητής
ΕΜΠ	ΕΜΠ	Χαροκόπειο Πανεπιστήμιο

Αθήνα
Νοέμβριος 2022

(Υπογραφή)

....
Παναγιώτης Δ. Χρυσομέρης
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Τα τελευταία χρόνια, η ανάπτυξη των εφαρμογών που χρησιμοποιούν Τεχνητή Νοημοσύνη (AI) σημειώνει ραγδαία αύξηση. Για να ικανοποιηθεί η αυξανόμενη ζήτηση για εφαρμογές μηχανικής μάθησης (ML), οι πάροχοι υπηρεσιών Cloud προσφέρουν συστήματα παροχής ML συμπερασμάτων (inference) ως διαδικτυακές υπηρεσίες (ML-as-a-Service), στις οποίες οι τελικοί χρήστες μπορούν να υποβάλουν αιτήματα για να επωφεληθούν από έτοιμες λύσεις ML, χωρίς να χρειάζεται να εγκαταστήσουν λογισμικό ή να παρέχουν τους δικούς τους διακομιστές. Συνήθως, τα αιτήματα παροχής ML inference συνοδεύονται από απαιτήσεις απόδοσης, γνωστές και ως περιορισμοί ποιότητας υπηρεσίας (QoS), στόχοι (SLOs) ή συμφωνίες (SLAs) επιπέδου υπηρεσίας, που ορίζονται από την εκάστοτε εφαρμογή. Συνεπώς, μια βασική πρόκληση για τους παρόχους υπηρεσιών Cloud είναι να εγγυηθούν τέτοιες απαιτήσεις μεγιστοποιώντας ταυτόχρονα την αποδοτικότητα της χρήσης των πόρων των υποδομών τους, καταλήγοντας σε μειωμένο λειτουργικό κόστος. Ωστόσο, η ικανοποίηση τέτοιων αντιφατικών στόχων βελτιστοποίησης καθίσταται ιδιαίτερα δυσχερής λόγω i) της υψηλής ποικιλομορφίας των διαθέσιμων λύσεων παροχής ML inference και ii) της μεταβλητότητας της απόδοσης λόγω της μη προβλεψιμότητας των αιτημάτων των χρηστών μέσα στη μέρα. Επιπλέον, οι πάροχοι υπηρεσιών Cloud συχνά τοποθετούν τις εφαρμογές σε κοντινούς κοινόχρηστους φυσικούς διακομιστές για να μεγιστοποιήσουν τη χρήση πόρων της υποδομής τους, κάτι που, ωστόσο, οδηγεί σε υποβάθμιση της απόδοσης λόγω των επιπτώσεων παρεμβολής στους πόρους.
Στην παρούσα εργασία, προτείνουμε ένα πλαίσιο προγνωστικής χρονοδρομολόγησης για μηχανές ML inference με επίγνωση των παρεμβολών και των πόρων, το οποίο είναι ικανό να χρησιμοποιεί αποτελεσματικά τους πόρους της KME (CPU) για την ικανοποίηση περιορισμών QoS. Το πλαίσιο μας λαμβάνει υπ’ όψη τον τρέχοντα φόρτο εργασίας και τη χρήση πόρων του Cloud αξιοποιώντας μετρήσεις συστήματος χαμηλού επιπέδου για να προβλέψει τα αιτήματα ανά δευτερόλεπτο (QPS) που θα επιτύχει μια μηχανή ML inference, καθώς και για να επιλέξει το κατάλληλο επίπεδο παραλληλισμού για την εκτέλεση. Παρουσιάζουμε, επίσης, μια προσέγγιση χωρίς μοντέλο για το πλαίσιο χρονοδρομολόγησης, η οποία πλοηγείται στον χώρο συμβιβασμού μεταξύ διαφορετικών παραλλαγών ML μοντέλων για μια ML inference εργασία, εκ μέρους των προγραμματιστών, για την επίτευξη του στόχου της εκάστοτε εφαρμογής με ελάχιστη χρήση πόρων. Ενσωματώνουμε τη λύση μας με τον Κυβερνήτη (Kubernetes), έναν από τους πιο ευρέως χρησιμοποιούμενους ενορχηστρωτές Cloud υπολογιστικών συστημάτων. Αξιολογούμε το πλαίσιο χρονοδρομολόγησης χρησιμοποιώντας ένα σύνολο ML inference μηχανών από τη σουίτα MLPerf Inference Benchmark. Τα πειραματικά αποτελέσματα δείχνουν ότι το πλαίσιο μας κάνει μέτρια χρήση των πόρων CPU, ανάλογα με το στόχο QoS και το φόρτο εργασίας στους πόρους, για να παραβιάσει τους περιορισμούς QoS, κατά μέσο όρο, 1.8x/3.1x λιγότερο συχνά, σε σύγκριση με το σύστημα παροχής ML inference μέγιστης/ελάχιστης χρήσης CPU αντίστοιχα, και με μεταβλητότητα απόδοσης που συγκεντρώνεται καλύτερα γύρω από το στόχο QoS, σε μια ποικιλία σεναρίων παρεμβολών και με διαφορετικούς περιορισμούς QoS. Επιπλέον, το πλαίσιο χωρίς μοντέλο σημειώνει, κατά μέσο όρο, 1.5x λιγότερες παραβιάσεις στους περιορισμούς QoS και 1.4x μικρότερη χρήση της CPU, σε σύγκριση με το πλαίσιο χρονοδρομολόγησης με συγκεκριμένο μοντέλο.

Λέξεις κλειδιά – Cloud, inference, χρονοδρομολόγηση, παρεμβολές, διαχείριση πόρων, KME, QoS, QPS, απόδοση, πρόβλεψη, Μηχανική Μάθηση, ταξινόμηση εικόνας, ανίχνευση αντικειμένων, Κυβερνήτης, MLPerf, iBench

Abstract

Over the last years, the growth of applications that utilize Artificial Intelligence (AI) is rapidly increasing and is expected to grow further in the future. To satisfy this ever-increasing demand for Machine Learning (ML) driven applications, Cloud providers offer inference serving systems as online services (ML-as-a-Service), which end-users can query to take advantage of "out-of-the-box" ML solutions without having to install software or provision their own servers. Typically, ML inference serving requests are accompanied with performance requirements, also known as Quality-of-Service (QoS) constraints, Service-Level-Objectives (SLOs) or Service-Level Agreements (SLAs), which correspond to latency constraints set by the respective application. To this end, a key challenge for Cloud providers is to guarantee such requirements while also maximizing the resource efficiency of their infrastructures, thus leading to reduced operational costs. However, satisfying such contradictory optimization goals becomes really challenging due to i) the high diversity in terms of ML inference serving solutions available and ii) the performance variability due to the unpredictability of user requests during the day. On top of that, Cloud providers tend to co-locate applications in shared physical servers to maximize the resource utilization of their infrastructure, which, however, imposes performance degradation due to resource interference effects.

In this diploma thesis, we propose an interference and resource aware, predictive scheduling framework for ML inference engines, that is capable of efficiently utilizing CPU resources to satisfy QoS constraints. Our framework considers the effect of resource interference in the Cloud by leveraging low-level system metrics to predict the Queries per Second (QPS) that an inference engine will achieve, based on the current load and resource utilization, as well as to select the appropriate parallelism level for deployment. We also introduce a model-less approach to the scheduling framework, which navigates the trade-off space of diverse ML model-variants for a specific inference task, on behalf of developers, to meet the application-specific objective with minimum resource utilization. We integrate our solution with Kubernetes, one of the most widely used cloud orchestration frameworks nowadays.

We evaluate our scheduling framework using a set of inference engines from the MLPerf Inference Benchmark Suite. Experimental results show that our scheduling framework utilizes a moderate amount of CPU resources, dependent on the target QoS and resource load, to violate QoS constraints, on average, 1.8x less often, compared to the max CPU utilization inference serving system, and 3.1x less often, compared to the min CPU utilization inference serving system, and with a performance variability that is better concentrated around the target QoS, for a variety of interference scenarios and different QoS constraints. Moreover, as the QoS constraints change, the model-less scheduling framework retains a similar, on average, overall performance and resource utilization, to the best performing, most efficient inference engine each time, resulting, on average, in 1.5x less violations of the QoS constraints and 1.4x less CPU utilization, compared to the model-specific scheduling framework.

Keywords – Cloud, inference, scheduling, interference-aware, resource management, CPU, QoS, QPS, throughput, performance, predictive, Machine Learning, image classification, object detection, model-less, Kubernetes, MLPerf, iBench

Ευχαριστίες

Αρχικά θα ήθελα να εκφράσω την ευγνωμοσύνη μου στον επιβλέποντα μου, καθηγητή Δημήτριο Σούντρη, για την εμπιστοσύνη και την ευκαιρία που μου έδωσε να εκπονήσω τη διπλωματική μου εργασία στο Εργαστήριο Μικροϋπολογιστών και Ψηφιακών Συστημάτων (MicroLab) στο ΕΜΠ.

Επίσης, θα ήθελα να ευχαριστήσω τους υποψήφιους διδάκτορες Δημοσθένη Μασούρο και Άγγελο Φερίκογλου για τη συνεργασία και την βοήθειά τους καθ’ όλη τη διάρκεια της διπλωματικής μου. Η συνεχής τριβή μας κατά τη διάρκεια της διπλωματικής με βοήθησε να αποκτήσω γρηγορότερα γνώσεις σχετικές με το αντικείμενο που εξετάσαμε, οι οποίες ταυτόχρονα είναι εφαρμόσιμες σε πολλούς τομείς της σύγχρονης τεχνολογίας και των συστημάτων υπολογιστών. Παράλληλα με εισήγαγαν στην ερευνητική προσέγγιση και εργασία.

Επιπλέον, θα ήθελα να ευχαριστήσω τους γονείς μου, τον αδερφό μου και τους φίλους μου για την συνεχή υποστήριξη τους σε ό,τι και αν επιχειρούσα κατά τη διάρκεια της ζωής και των σπουδών μου, πράγμα που μου έδινε δύναμη να συνεχίσω να επιδιώκω τους στόχους μου. Τέλος, ένα μεγάλο ευχαριστώ στην σύντροφό μου, η οποία με υπομονή με στήριξε και μου συμπαραστάθηκε στις δύσκολες στιγμές.

Contents

Contents

Περίληψη 5

Abstract 7

Ευχαριστίες 9

Contents 11

Εκτεταμένη Ελληνική Περίληψη 15

1 Εισαγωγή 15

2 Θεωρητικό Υπόβαθρο 17

2.1 Κυβερνήτης (Kubernetes) και Ενορχήστρωση Containers 17

2.2 MLPerf™ Inference Benchmark Σουίτα 17

2.3 iBench: Ποσοτικοποίηση παρεμβολών για εφαρμογές σε κέντρα δεδομένων 19

2.4 Μηχανική Μάθηση (ML) 19

3 Χαρακτηρισμός των MLPerf Inference Benchmarks 20

3.1 Πειραματικό Περιβάλλον 20

3.2 Σενάριο Ενιαίας Ροής 21

3.3 Σενάριο Πολλαπλών Ροών 23

4 Χρονοδρομολογητής 26

4.1 Εκτός Σύνδεσης: Εκπαίδευση των Regression Μοντέλων Μηχανικής Μάθησης 26

4.2 Σε Σύνδεση: Προγνωστικός Χρονοδρομολογητής Inference Μηχανών με Επίγνωση των Παρεμβολών και των Πόρων 29

5 Αξιολόγηση 31

5.1 Περιγραφή Σεναρίων 31

5.2 Περιγραφή Πειραμάτων 31

5.3 Αποτελέσματα Χρονοδρομολογητή Inference Μηχανών Με Συγκεκριμένο Μοντέλο & Σύγκριση Χρονοδρομολογητών 32

5.4 Αποτελέσματα Χρονοδρομολογητή Inference Μηχανών Χωρίς Μοντέλο & Σύγκριση Χρονοδρομολογητών 34

6 Σύνοψη 37

1 Introduction 39

2 Related Work 43

2.1 INFaaS: Automated Model-less Inference Serving 43

2.2 Serving Heterogeneous Machine Learning Models on Multi-GPU Servers with Spatio-Temporal Sharing 43

2.3 Deep Learning Workload Scheduling in GPU Datacenters: Taxonomy, Challenges and Vision 44

2.4 Morphling: Fast, Near-Optimal Auto-Configuration for Cloud-Native Model Serving 44
3 Background 47
3.1 Kubernetes, a Container Orchestrator 47
3.1.1 Docker containers and Orchestration 47
3.1.2 Kubernetes Master Node(s) Components 50
3.1.3 Kubernetes Worker Node(s) Components 50
3.1.4 Kubernetes Architecture 51
3.1.5 Kubernetes Resources 54
3.1.6 Kubernetes Scheduling 56
3.2 MLPerf™ Inference Benchmark Suite 56
3.2.1 MLPerf Inference Benchmarks for Image Classification and Object Detection Tasks 57
3.3 iBench: Quantifying interference for datacenter applications 60
3.4 Machine Learning (ML) 61
3.4.1 Regression Models in Machine Learning 62
4 Characterization 77
4.1 Experimental Infrastructure 77
4.1.1 System Setup 77
4.1.2 MLPerf Inference Benchmarks for Image Classification and Object Detection Tasks 77
4.1.3 iBench 78
4.2 SingleStream Benchmark Scenario 78
4.2.1 With interference 78
4.2.2 Without interference 80
4.2.3 CPU Limits 82
4.3 MultiStream Benchmark Scenario 83
4.3.1 Onnxruntime framework - without interference 83
4.3.2 Onnxruntime framework - with interference 84
4.3.3 Tensorflow framework - without interference 87
4.3.4 Tensorflow framework - with interference 89
5 Scheduler 93
5.1 Offline: Training of the Machine Learning Regression Models 93
5.2 Online: Interference and resource aware predictive inference engine scheduler 100
5.2.1 Model-Specific Deployment 100
5.2.2 Model-less approach 102
6 Evaluation 103
6.1 Scenarios Description 103
6.2 Experiments Description 104

6.2.1 Model-Specific Inference Engine Scheduler 104
6.2.2 Model-less Inference Engine Scheduler 105
6.3 Model-Specific Inference Engine Scheduler Results & Scheduler Comparison 106
6.3.1 Violations of the QoS constraint 106
6.3.2 Resources Utilization 113
6.3.3 Performance Distribution 118
6.4 Model-less Inference Engine Scheduler Results & Scheduler Comparison 124
6.4.1 Violations of the QoS constraint 124
6.4.2 Resources Utilization 126
6.4.3 Performance Distribution 128
7 Conclusion 131
Bibliography 133

Εκτεταμένη Ελληνική Περίληψη

1 Εισαγωγή

Ο αριθμός των εφαρμογών που βασίζονται σε inference από μοντέλα Μηχανικής Μάθησης (ML) είναι ήδη μεγάλος και αναμένεται να συνεχίσει να αυξάνεται. Το Facebook, για παράδειγμα, εξυπηρετεί δεκάδες τρισεκατομμύρια αιτήματα ML συμπερασμάτων (inference) την ημέρα. Αυτά τα αιτήματα ML inference συνήθως εκφορτώνονται στο Cloud. Η αυξανόμενη αποτελεσματικότητα της Μηχανικής Μάθησης (ML) και η έλευση των υπηρεσιών Cloud έχει φέρει ταχεία ανάπτυξη στα συστήματα παροχής ML inference ως διαδικτυακές υπηρεσίες (MLaaS) όπως το Google Cloud Vertex AI ή το Amazon Elastic Inference, όπου οι πάροχοι υπηρεσιών Cloud προσφέρουν τα συστήματα παροχής ML inference τους.

Γενικά, ένας κύκλος ζωής ML έχει δύο διακριτές φάσεις - εκπαίδευση και συμπέρασμα (inference). Σε μια τυπική ροή εργασίας MLaaS, οι προγραμματιστές σχεδιάζουν και εκπαιδεύουν μοντέλα ML εκτός σύνδεσης. Η φάση εκπαίδευσης συνήθως χαρακτηρίζεται από μεγάλα σύνολα δεδομένων, μακροχρόνιες αναζητήσεις υπερπαραμέτρων, αποκλειστική χρήση πόρων υλικού και χωρίς προθεσμίες ολοκλήρωσης. Τα εκπαιδευμένα μοντέλα δημοσιεύονται στη συνέχεια στο Cloud για την παροχή διαδικτυακών υπηρεσιών ML inference (φάση inference), που συνήθως εκτελούνται σε πακέτα (containers), και μπορούν να υποβληθούν σε αιτήσεις από διάφορες εφαρμογές τελικού χρήστη ώστε να κάνουν προβλέψεις για δεδομένες εισόδους. Η παροχή ML inference απαιτεί αποδοτικά ως προς το κόστος συστήματα που προσφέρουν προβλέψεις με περιορισμούς καθυστέρησης ή ακρίβειας, ενώ χειρίζονται απρόβλεπτες και εκρηκτικές αφίξεις αιτημάτων.

Τα συστήματα παροχής ML inference αντιμετωπίζουν μια σειρά από προκλήσεις λόγω των α) διαφορετικών απαιτήσεων μεταξύ των εφαρμογών, β) των διαφορετικών παραλλαγών των μοντέλων για την ίδια ML εργασία και γ) των κυμαινόμενων παρεμβολών στους πόρους του συστήματος. Οι εφαρμογές στέλνουν αιτήματα που διαφέρουν ως προς την απόδοση, την καθυστέρηση, το κόστος, την ακρίβεια. Για παράδειγμα, για αιτήματα στο ίδιο μοντέλο αναγνώρισης αντικειμένων, ένα αυτόνομο όχημα, το οποίο λαμβάνει συνεχώς δεδομένα από πολλές κάμερες, απαιτεί ML inference σε πραγματικό χρόνο, ενώ μια μηχανή γραμμής παραγωγής που εκτελεί επιθεώρηση υφής ή ανίχνευση ελαττωμάτων, απαιτεί κυρίως ακρίβεια και μετέπειτα ελαχιστοποίηση της καθυστέρησης. Επίσης, οι διάφορες παραλλαγές των μοντέλων δημιουργούν ένα χώρο συμβιβασμού μεταξύ της απόδοσης, της καθυστέρησης, της χρήσης της μνήμης ή της ακρίβειας. Επιπρόσθετα, ανάλογα το πλήθος των εφαρμογών που υποβάλλουν αιτήματα σε μια μηχανή ML inference, ενδέχεται να υπάρχει λιγότερη ή μεγαλύτερη συμφόρηση στους κοινόχρηστους πόρους των διακομιστών Cloud. Επιπλέον, οι πάροχοι υπηρεσιών Cloud τείνουν να τοποθετούν τις εφαρμογές σε κοντινούς κοινόχρηστους φυσικούς διακομιστές για να μεγιστοποιήσουν τη χρήση πόρων της υποδομής τους και να μειώσουν το λειτουργικό τους κόστος, γεγονός που μπορεί να αυξήσει περαιτέρω τη συμφόρηση στους κοινόχρηστους πόρους.

Ως αποτέλεσμα, μαζί αυτοί οι παράγοντες καθιστούν δύσκολο για τους παρόχους υπηρεσιών Cloud να εγγυηθούν έναν περιορισμό ποιότητας υπηρεσίας (QoS) σε μια εφαρμογή για κάθε αίτημα ML inference, με παράλληλη μεγιστοποίηση της αποδοτικότητας της χρήσης των πόρων των υποδομών τους. Απλές προσεγγίσεις χρονοδρομολόγησης που δρομολογούν μηχανές ML inference στο Cloud να εκτελεστούν με έναν καθορισμένο αριθμό διαθέσιμων πόρων, χωρίς να λαμβάνουν υπ' όψη τη μη προβλεψιμότητα του τύπου και του όγκου των αιτημάτων των χρηστών κατά τη διάρκεια της ημέρας και του φόρτου εργασίας που θα μπορούσαν να δημιουργήσουν στους πόρους της υποδομής του Cloud, ενδέχεται να έχουν ως αποτέλεσμα παροχή υπερβολικά πολλών πόρων στην εφαρμογή, το οποίο οδηγεί σε αυξημένο λειτουργικό κόστος ή παροχή λιγότερων πόρων από όσα χρειάζεται η εφαρμογή, κάτι που οδηγεί σε περισσότερες παραβιάσεις QoS των εφαρμογών. Επίσης, η αξιοποίηση της πληθώρας παραλλαγών των ML inference μοντέλων απαιτεί καλή κατανόηση των πλαισίων υποστήριξης, των βελτιστοποιητών γραφημάτων μοντέλων και των αρχιτεκτονικών υλικού τους, κάτι το οποίο περιορίζει τον τελικό αριθμό των μοντέλων που μπορούν να αξιοποιήσουν οι προγραμματιστές για την καλύτερη κάλυψη των διαφορετικών απαιτήσεων κάθε εφαρμογής.

Για να αντιμετωπίσουμε τις αναδυόμενες προκλήσεις της χρονοδρομολόγησης ML inference, σε αυτή τη διπλωματική εργασία, σχεδιάζουμε και εφαρμόζουμε ένα πλαίσιο προγνωστικής χρονοδρομολόγησης με επίγνωση των παρεμβολών και των πόρων, για μηχανές ML inference, το οποίο είναι ικανό να χρησιμοποιεί αποτελεσματικά τους πόρους της Κεντρικής Μονάδας Επεξεργασίας (CPU) για την ικανοποίηση περιορισμών QoS. Το πλαίσιό μας λαμβάνει υπ' όψη την επίδραση της παρεμβολής στους πόρους του Cloud αξιοποιώντας μετρήσεις συστήματος χαμηλού επιπέδου για την εκπαίδευση και τη χρήση ενός μοντέλου ML Regression για την πρόβλεψη των αιτημάτων ανά δευτερόλεπτο (QPS) που θα επιτύχει μια μηχανή ML inference, με βάση τον τρέχοντα φόρτο εργασίας και χρήση πόρων. Το πλαίσιο χρονοδρομολόγησης μας στοχεύει να δρομολογήσει τη μηχανή ML inference να εκτελεστεί με ένα επίπεδο παραλληλισμού που έχει ως αποτέλεσμα μια απόδοση που πληροί τον απαιτούμενο περιορισμό QoS, ενώ κάνει όσο το δυνατόν μικρότερη χρήση πόρων για το σύστημα. Παρουσιάζουμε επίσης ένα πλαίσιο χρονοδρομολόγησης χωρίς μοντέλο το οποίο εισάγει μια διεπαφή χωρίς μοντέλο, όπου οι προγραμματιστές χρειάζεται να καθορίσουν μόνο την εργασία ML inference που θέλουν να εκτελέσουν (ταξινόμηση εικόνας, ανίχνευση αντικειμένων) και την απόδοση υψηλού επιπέδου που απαιτούν ως στόχο QoS . Το πλαίσιο χρονοδρομολόγησης χωρίς μοντέλο επιλέγει την καλύτερη Μηχανή ML Inference, από μια ομάδα εγγεγραμμένων, εκπαιδευμένων μοντέλων ML inference για τη συγκεκριμένη εργασία, με το κατάλληλο επίπεδο παραλληλισμού για την ολοκλήρωση της εργασίας με τον λιγότερο απαιτητικό σε πόρους τρόπο, ενώ στοχεύει να ικανοποιήσει τον περιορισμό QoS σύμφωνα με το τρέχον επίπεδο συνθηκών παρεμβολής στο σύστημα.

Δείχνουμε ότι το πλαίσιο χρονοδρομολόγησης μας κάνει μια μέτρια χρήση των πόρων CPU, ανάλογα με το στόχο QoS και το φόρτο εργασίας στους πόρους, για να παραβιάσει τους περιορισμούς QoS, κατά μέσο όρο, 1.8x λιγότερο συχνά, σε σύγκριση με το σύστημα παροχής ML inference μέγιστης χρήσης CPU και 3.1x λιγότερο συχνά, σε σύγκριση με το σύστημα παροχής ML inference ελάχιστης χρήσης CPU, και με μεταβλητότητα απόδοσης που συγκεντρώνεται καλύτερα γύρω από το στόχο QoS, σε μια ποικιλία σεναρίων παρεμβολών και με διαφορετικούς περιορισμούς QoS. Επιπλέον, καθώς αλλάζουν οι περιορισμοί QoS, το πλαίσιο χρονοδρομολόγησης χωρίς μοντέλο διατηρεί παρόμοια, κατά μέσο όρο, συνολική απόδοση και χρήση πόρων, με την καλύτερη σε απόδοση και πιο αποδοτική σε χρήση πόρων μηχανή ML inference κάθε φορά, με αποτέλεσμα, κατά μέσο όρο, 1.5x λιγότερες παραβιάσεις στους περιορισμούς QoS και 1.4x μικρότερη χρήση της CPU, σε σύγκριση με το πλαίσιο χρονοδρομολόγησης με συγκεκριμένο μοντέλο.

2 Θεωρητικό Υπόβαθρο

2.1 Κυβερνήτης (Kubernetes) και Ενορχήστρωση Containers

Πακέτο (Container): Το container είναι μια τυποποιημένη μονάδα λογισμικού η οποία συγκεντρώνει τον κώδικα, αλλά και όλες τις εξαρτίσεις του έτσι ώστε η εφαρμογή να μπορεί να εκτελείται γρήγορα και αξιόπιστα σε ποικίλα περιβάλλοντα υπολογιστών. Τα κέντρα δεδομένων σήμερα χρησιμοποιούν αυτή τη νέα τεχνολογία εικονικοποίησης, καθώς έχει πολλά πλεονεκτήματα συγκριτικά με τις εικονικές μηχανές. Ενδεικτικά αναφέρουμε την ευέλικτη δημιουργία και ανάπτυξη εφαρμογών, τη διευκόλυνση ενός γρηγορότερου κύκλου ανάπτυξης του λογισμικού, τη συνέπεια σχετικά με το περιβάλλον ανάπτυξης και την απομόνωση πόρων.

Ενορχήστρωση: Σε μεγάλες συστοιχίες υπολογιστών υπάρχει η ανάγκη ενορχήστρωσης και διαχείρισης των containers. Η ανάγκη αυτή καλύπτεται από υλοποιήσεις όπως αυτή του Κυβερνήτη, ενός έργου ανοιχτού λογισμικού που ξεκίνησε να αναπτύσσεται με πρωτοβουλία της Google. Ο Κυβερνήτης αποτελεί την πιο ευρέως χρησιμοποιούμενη υλοποίηση. Ξεκινώντας από το υψηλότερο επίπεδο αφαίρεσης, η αρχιτεκτονική του περιλαμβάνει έναν ή περισσότερους κόμβους «αφέντη» (master), οι οποίοι είναι το μυαλό του συστήματος, αποτελούν το πεδίο ελέγχου, παίρνουν αποφάσεις και αντιδρούν σε διάφορα γεγονότα που λαμβάνουν χώρα σε αυτό. Η άλλη ομάδα κόμβων είναι οι επονομαζόμενοι κόμβοι «εργάτες» (workers), στους οποίους αποστέλλονται και εκτελούνται όλες οι εργασίες-εφαρμογές. Ένας κόμβος master περιέχει: kube-apiserver, etcd, kube-scheduler, kube-controller-manager. Από την άλλη, ένας κόμβος worker περιέχει: kubelet, kube-proxy, container runtime.

Οι εφαρμογές, αφού τοποθετηθούν μέσα σε containers, τοποθετούνται σε Pods τα οποία μπορούν να περιέχουν ένα ή περισσότερα containers ή και μονάδες αποθήκευσης (volumes). Ο Κυβερνήτης υποστηρίζει μια πληθώρα υπηρεσιών και παρέχει ποικίλες δυνατότητες στους χρήστες και προγραμματιστές, ευνοώντας την αυτοματοποίηση την αναγκαίων εργασιών. Αρχικά, παρέχει την δυνατότητα χρήσης εργασιών (Jobs). Μια εργασία δημιουργεί ένα ή περισσότερα Pods και διασφαλίζει ότι ένας καθορισμένος αριθμός από αυτά τερματίζεται με επιτυχία. Επίσης, μας δίνει την δυνατότητα καθορισμού αιτημάτων (requests) και περιορισμών (limits) στη χρήση της κεντρικής μονάδας επεξεργασίας ή της κύριας μνήμης σε επίπεδο container. Μία άλλη παροχή είναι ο ενσωματωμένος χρονοδρομολογητής εργασιών. Ο τελευταίος βασιζόμενος σε μετρικές υψηλού επιπέδου, αφαιρετικές σε επίπεδο εικονικοποίησης, όπως η χρήση των επεξεργαστών και μνήμης, παίρνει αποφάσεις σχετικά με την τοποθέτηση των Pods. Η διαδικασία με την οποία οι αποφάσεις λαμβάνονται περιλαμβάνει δύο στάδια. Αρχικά, εξετάζονται όλοι οι υποψήφιοι κόμβοι σχετικά με τη διαθεσιμότητά τους, την ικανότητά τους να εξυπηρετήσουν την εισερχόμενη εφαρμογή. Στη συνέχεια όσοι από αυτούς κριθούν κατάλληλοι, βαθμολογούνται με τη χρήση μιας σειράς από συναρτήσεις αξιολόγησης.

2.2 MLPerf™ Inference Benchmark Σουίτα

Οι σουίτες MLPerf benchmark είναι το πρότυπο για τη μέτρηση της απόδοσης ενός συστήματος μηχανικής μάθησης. Κάθε σουίτα εστιάζει σε διαφορετικούς τύπους συστημάτων και φόρτους εργασίας. Το MLPerf Inference benchmark suite είναι μια σουίτα που αξιολογεί συστήματα ML inference, μετρώντας πόσο γρήγορα επεξεργάζονται εισόδους και παράγουν αποτελέσματα χρησιμοποιώντας/τρέχοντας ένα εκπαιδευμένο μοντέλο σε μια ποικιλία σεναρίων εκτέλεσης. Παραδείγματα εργασιών περιλαμβάνουν τη σύσταση, την απάντηση ερωτήσεων, την ομιλία σε κείμενο, την ανίχνευση αντικειμένων και την αναγνώριση εικόνας.

2.2.1 MLPerf Inference Benchmarks για Εργασίες Ταξινόμησης Εικόνων και Ανίχνευσης Αντικειμένων

Η όραση υπολογιστών είναι ένα πεδίο τεχνητής νοημοσύνης (AI) που επιτρέπει σε υπολογιστές και συστήματα να αντλούν σημαντικές πληροφορίες από ψηφιακές εικόνες, βίντεο και άλλες οπτικές εισροές — και να προβαίνουν σε ενέργειες ή να κάνουν συστάσεις με βάση αυτές τις πληροφορίες. Σε αυτό το πεδίο, δύο σημαντικές εργασίες για την αξιολόγηση της απόδοσης ενός συστήματος ML inference, χρησιμοποιώντας τη σουίτα MLPerf Inference benchmark, είναι η ταξινόμηση εικόνων και η ανίχνευση αντικειμένων.

Παρακάτω παρουσιάζεται η υλοποίηση αναφοράς για τα MLPerf Inference benchmarks. Κάθε benchmark ορίζεται από ένα προπαιδευμένο μοντέλο, ένα πλαίσιο υποστήριξης και ένα σύνολο δεδομένων.

Area	Task	**model**	**framework**	**accuracy**	**dataset**	**precision**
Vision	Image Classification	resnet50-v1.5	tensorflow	76.456%	imagenet2012 validation	fp32
Vision	Image Classification	resnet50-v1.5	onnx	76.456%	imagenet2012 validation	fp32
Vision	Image Classification	mobilenet-v1	tensorflow	71.676%	imagenet2012 validation	fp32
Vision	Image Classification	mobilenet-v1 quantized	tensorflow	70.694%	imagenet2012 validation	int8
Vision	Image Classification	mobilenet-v1	tflite	71.676%	imagenet2012 validation	fp32
Vision	Image Classification	mobilenet-v1	onnx	71.676%	imagenet2012 validation	fp32
Vision	Object Detection	ssd-mobilenet 300x300	tensorflow	mAP 0.23	coco resized to 300x300	fp32
Vision	Object Detection	ssd-mobilenet 300x300 quantized finetuned	tensorflow	mAP 0.23594	coco resized to 300x300	int8
Vision	Object Detection	ssd-mobilenet 300x300 symmetrically quantized finetuned	tensorflow	mAP 0.234	coco resized to 300x300	int8
Vision	Object Detection	ssd-mobilenet 300x300	onnx	mAP 0.23	coco resized to 300x300	fp32

Πίνακας 2.1: Υποστηριζόμενα μοντέλα της σουίτας MLPerf Inference Benchmark Suite για ML εργασίες ταξινόμησης εικόνων και ανίχνευσης αντικειμένων

Το MLPerf έχει ορίσει τέσσερα διαφορετικά inference σενάρια που μιμούνται τον φόρτο εργασίας διαφορετικών εφαρμογών: ενιαία ροή (SingleStream), πολλαπλές ροές (MultiStream), διακομιστής (Server), εκτός σύνδεσης (Offline). Στα σενάρια ενιαίας ροής και πολλαπλών ροών που θα ασχοληθούμε, το κάθε inference αίτημα στέλνεται αφού ολοκληρωθεί το προηγούμενο, με 1 δείγμα/αίτημα ή 8 δείγματα/αίτημα αντίστοιχα.

2.3 iBench: Ποσοτικοποίηση παρεμβολών για εφαρμογές σε κέντρα δεδομένων

Οι παρεμβολές μεταξύ εφαρμογών είναι ένας από τους κύριους λόγους που αναγκάζουν τα σύγχρονα κέντρα δεδομένων (DCs) να λειτουργούν με χαμηλή χρήση της υποδομής τους. Το iBench workload suite είναι μια σουίτα φόρτου εργασίας που βοηθά στην ποσοτικοποίηση της πίεσης που ασκούν διαφορετικές εφαρμογές σε διάφορους κοινόχρηστους πόρους, και παρομοίως της πίεσης που μπορούν να ανεχθούν σε αυτούς τους πόρους, με αποτέλεσμα σημαντικές βελτιώσεις απόδοσης ή/και αποδοτικότητας στις εφαρμογές. Το iBench αποτελείται από ένα σύνολο προσεκτικά κατασκευασμένων benchmarks που προκαλούν παρεμβολές ρυθμιζόμενης, αυξανόμενης έντασης σε κοινόχρηστους πόρους όπως οι πυρήνες της ΚΜΕ (CPU), η ιεραρχία της κρυφής μνήμης (L2 cache, L3 cache), το εύρος ζώνης (memory bandwidth) και η χωρητικότητα μνήμης (memory capacity). Κάθε φόρτος εργασίας του iBench (πηγή παρεμβολών (SoI)) ασκεί πίεση σε έναν συγκεκριμένο κοινόχρηστο πόρο και οι iBench πηγές παρεμβολών έχουν σχεδιαστεί έτσι ώστε να μην υπάρχει αλληλοσυγκάλυψη στην επίδρασή τους στους πόρους του συστήματος.

2.4 Μηχανική Μάθηση (ML)

Η μηχανική μάθηση (ML) είναι ένα πεδίο έρευνας, που ανήκει στην τεχνητή νοημοσύνη (AI), αφιερωμένο στην κατανόηση και τη δημιουργία μεθόδων που «μαθαίνουν», δηλαδή μέθοδοι που αξιοποιούν δεδομένα για τη βελτίωση της απόδοσης σε κάποιο σύνολο εργασιών. Οι αλγόριθμοι μηχανικής μάθησης δημιουργούν ένα μοντέλο που βασίζεται σε δείγματα δεδομένων, γνωστά ως δεδομένα εκπαίδευσης, προκειμένου να κάνουν ακριβείς προβλέψεις για τα αποτελέσματα ή να πάρουν αποφάσεις χωρίς να έχουν προγραμματιστεί ρητά για να το κάνουν. Υπάρχουν τρεις προσεγγίσεις μηχανικής μάθησης:

Εποπτευόμενη μάθηση: Οι αλγόριθμοι εποπτευόμενης μάθησης δημιουργούν ένα μαθηματικό μοντέλο ενός συνόλου δεδομένων εκπαίδευσης που περιέχει εισόδους και τις επιθυμητές εξόδους τους (ονομάζεται επίσης εποπτικό σήμα), με στόχο την εκμάθηση ενός γενικού κανόνα, μιας συνάρτησης, που αντιστοιχεί διανύσματα (εισόδους) σε ετικέτες (εξόδους), με βάση παραδείγματα ζευγών εισόδου-εξόδου. Αυτή η συνάρτηση που προκύπτει μπορεί στη συνέχεια να χρησιμοποιηθεί για την αντιστοίχιση νέων παραδειγμάτων. Σε αυτή τη κατηγορία ανήκουν η ταξινόμηση (classification) και το regression.

Μη εποπτευόμενη μάθηση: Οι αλγόριθμοι μάθησης χωρίς επίβλεψη λαμβάνουν ένα σύνολο δεδομένων που περιέχει μόνο εισόδους, που σημαίνει ότι δεν δίνονται ετικέτες, αφήνοντάς τους χωρίς επίβλεψη να εντοπίζουν κοινά σημεία στα δεδομένα και να αντιδρούν με βάση την παρουσία ή την απουσία τέτοιων κοινών στοιχείων σε κάθε νέο κομμάτι δεδομένων. Η μάθηση χωρίς επίβλεψη μπορεί να είναι ένας στόχος από μόνος του (ανακάλυψη κρυφών μοτίβων στα δεδομένα) ή ένα μέσο για την επίτευξη ενός σκοπού (εκμάθηση χαρακτηριστικών).

Ενισχυτική μάθηση: Η ενισχυτική μάθηση ασχολείται με το πώς ένα πρόγραμμα με συγκεκριμένο στόχο πρέπει να περιηγείται στον δυναμικό χώρο προβλημάτων, ώστε μέσω ανατροφοδότησης να μεγιστοποιήσει την ανταμοιβή του.

2.4.1 Μοντέλα Regression στη Μηχανική Μάθηση

Στη στατιστική μοντελοποίηση, η Regression ανάλυση είναι ένα σύνολο στατιστικών διαδικασιών για την εκτίμηση των σχέσεων μεταξύ μιας εξαρτημένης μεταβλητής (συχνά αποκαλούμενη μεταβλητή «αποτέλεσμα» ή «απόκριση» ή «ετικέτα» στη γλώσσα μηχανικής μάθησης) και μία ή περισσότερες ανεξάρτητες μεταβλητές (συχνά αποκαλούνται «προγνωστικοί παράγοντες», «συμμεταβλητές», «επεξηγηματικές μεταβλητές» ή «χαρακτηριστικά»). Στον τομέα της μηχανικής μάθησης, το Regression είναι μια εποπτευόμενη τεχνική μάθησης που βοηθά στην εύρεση της συσχέτισης μεταξύ των μεταβλητών και μας δίνει τη δυνατότητα να προβλέψουμε τη συνεχή, αριθμητική μεταβλητή εξόδου με βάση τη μία ή περισσότερες ανεξάρτητες μεταβλητές εισόδου. Μερικοί αλγόριθμοι Regression ανάλυσης είναι οι εξής:

Γραμμικοί: Linear Regression, Ridge regression, Lasso Regression, Bayesian Linear Regression, Stochastic Gradient Descent (SGD), Support Vector Regression (SVR)

Μη γραμμικοί: K-nearest Neighbors Regression, Gaussian process regression (GPR), Decision Tree Regression, Random Forest Regression, Multi-layer Perceptron regression, XGBoost Regression.

3 Χαρακτηρισμός των MLPerf Inference Benchmarks

3.1 Πειραματικό Περιβάλλον

Το σύστημα που δημιουργήθηκε για τον χαρακτηρισμό των MLPerf Inference Benchmarks και την μετέπειτα αξιολόγηση του χρονοδρομολογητή μας με την χρήση αυτών, απαρτίζεται από δύο εικονικές μηχανές, οι οποίες αποτελούν το σύμπλεγμα (cluster) πάνω στο οποίο έχει στηθεί ο Κυβερνήτης (Kubernetes). Ο ένας κόμβος λειτουργεί ως κύριος κόμβος (master) με 4 πυρήνες ΚΜΕ (CPU) και 8 GB κύριας μνήμης, και ο άλλος ως κόμβος «εργάτης» (worker) με 8 πυρήνες ΚΜΕ (CPU) και 16 GB κύριας μνήμης.

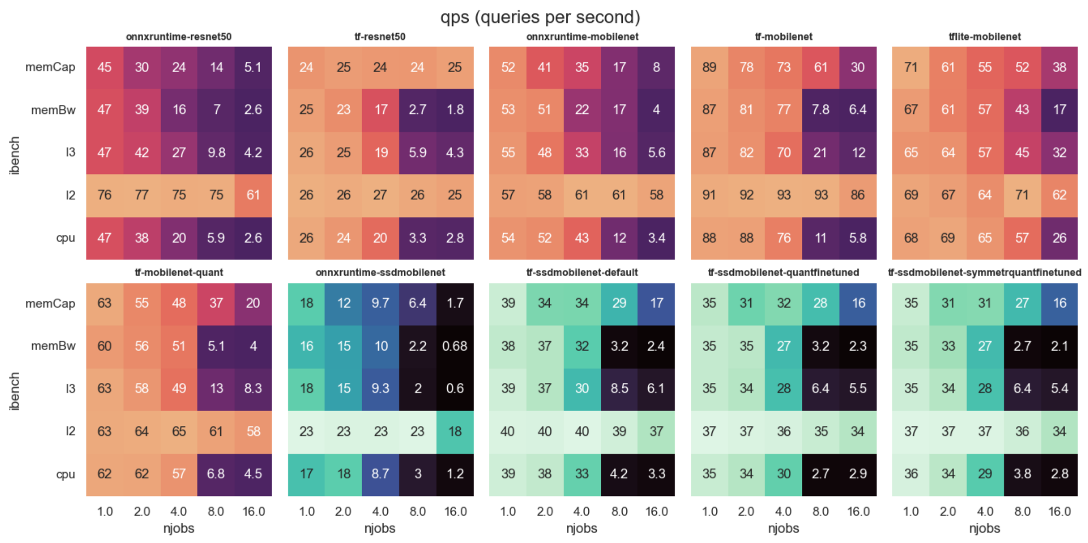
Virtual Machines		
VM-name	**Cores**	**RAM (GB)**
master	4	8
worker	8	16

Πίνακας 3.1: Χαρακτηριστικά εικονικών μηχανών

Τα MLPerf Inference Benchmarks, καθώς και τα iBench benchmarks τρέχουν ως Pods, ελεγχόμενα από Jobs, στο σύμπλεγμά μας.

3.2 Σενάριο Ενιαίας Ροής

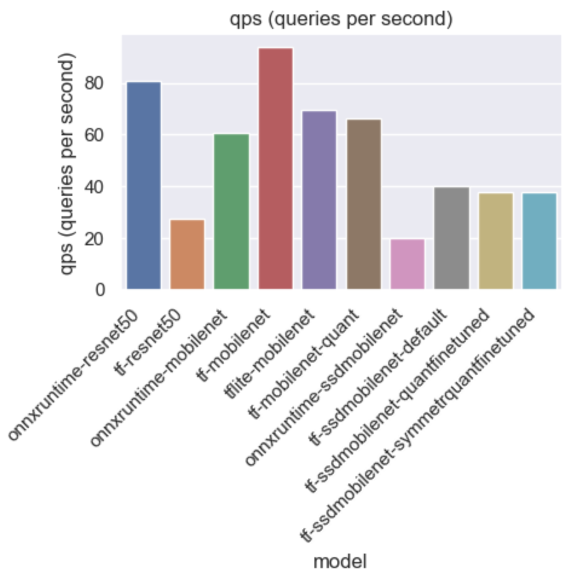
3.2.1 Με Παρεμβολές



Σχήμα 3.1: Μέτρηση QPS (έγχρωμα τετράγωνα) των MLPerf Inference benchmarks για διάφορες πηγές παρεμβολών «iBench» σε διαφορετικές εντάσεις (njobs).

Πρώτον, βλέπουμε ότι γενικά, όσο περισσότερα «iBench» πιέζουν ένα κοινόχρηστο πόρο, δηλαδή όσο πιο έντονα πιέζεται ο πόρος, τόσο πιο πολύ μειώνεται το QPS του μοντέλου. Ωστόσο, στη κρυφή μνήμη L2, με έως και 16 ταυτόχρονες πηγές παρεμβολής "iBench" να την πιέζουν, δεν υπάρχει σημαντική επίδραση στο QPS που πετυχαίνουν τα μοντέλα. Αυτό σημαίνει ότι τα μοντέλα μας μπορούν να αντέξουν υπερβολική πίεση στην κρυφή μνήμη L2 πριν αρχίσει να μειώνεται η απόδοσή τους. Δεύτερον, βλέπουμε ότι ενώ όλα τα μοντέλα είναι επιρρεπή σε μείωση της απόδοσης καθώς αυξάνεται η πίεση στη χωρητικότητα της μνήμης, αυτό δεν ισχύει για το μοντέλο tf-resnet50. Τρίτον, παρατηρούμε ότι το tflite-mobilenet καταφέρνει να διατηρήσει το QPS του υψηλότερα από τα άλλα μοντέλα όταν αυξάνονται οι πηγές παρεμβολής στους κοινόχρηστους πόρους, καθώς παρατηρούμε τη χαμηλότερη ποσοστιαία πτώση απόδοσης σε σύγκριση με τα άλλα μοντέλα. Για παράδειγμα, για αύξηση πίεσης στην χωρητικότητα της μνήμης από 1 σε 16 πηγές παρεμβολής, το QPS του tflite-mobilenet πέφτει στο μισό, ενώ το QPS του mobilenet με πλαίσιο υποστήριξης ONNX πέφτει στο ένα έβδομο.

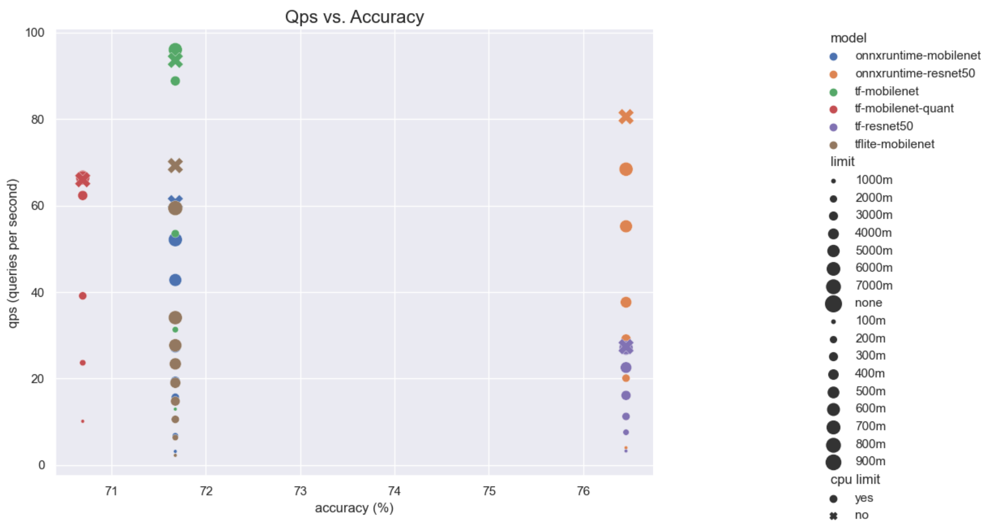
3.2.2 Χωρίς Παρεμβολές



Σχήμα 3.2: Μέτρηση απόδοσης (QPS) των MLPerf Inference benchmarks στο σενάριο ενιαίας ροής. Τα πρώτα έξι benchmarks εκτελούν μια εργασία ταξινόμησης εικόνων, ενώ τα τελευταία τέσσερα εκτελούν μια εργασία ανίχνευσης αντικειμένων.

Παρατηρούμε ότι για την εργασία ταξινόμησης εικόνων, επιτυγχάνουμε την καλύτερη απόδοση με το mobilenet με πλαίσιο υποστήριξης tensorflow στα 93 QPS, με δεύτερο καλύτερο το onnxruntime-resnet50 με διαφορά 13% και την χειρότερη απόδοση με το tf-resnet50. Αντίθετα, στην εργασία ανίχνευσης αντικειμένων, το benchmark με την υψηλότερη απόδοση είναι το tf-ssd-mobilenet-default, στα 40 QPS, με διαφορά μικρότερη του 6% από τα υπόλοιπα ssd-mobilenet με tensorflow, με το onnxruntime-ssd-mobilenet να πετυχαίνει το μικρότερο QPS.

3.2.3 Με Περιορισμούς στην Χρήση ΚΜΕ (CPU)



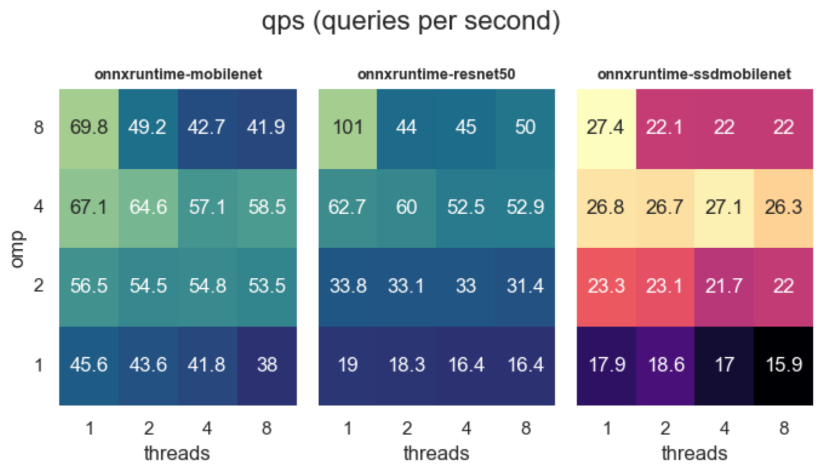
Σχήμα 3.3: QPS έναντι Ακρίβειας των MLPerf Inference benchmarks που εκτελούν μια εργασία ταξινόμησης εικόνας, με διάφορα επιβαλλόμενα όρια στην χρήση ΚΜΕ (CPU).

Μετράμε το επιτυγχανόμενο QPS κάθε μοντέλου MLPerf Inference, ενώ παρεμποδίζουμε με ελεγχόμενο τρόπο την απόδοσή τους θέτοντας περιορισμούς στη χρήση ΚΜΕ (CPU). Από το παραπάνω σχήμα, είναι σαφές ότι ενώ το μοντέλο tf-mobilenet πετυχαίνει το υψηλότερο QPS, το μοντέλο onnxruntime-resnet50 έχει μεγαλύτερο ποσοστό ακρίβειας, με λίγο χαμηλότερη απόδοση. Επίσης, παρατηρούμε ότι τα μοντέλα με πλαίσιο υποστήριξης tensorflow επιτυγχάνουν QPS κοντά στο μέγιστο για μικρούς περιορισμούς στην CPU, ενώ τα υπόλοιπα μοντέλα κάνουν μεγάλα άλματα στην απόδοση με κάθε μείωση στους περιορισμούς.

3.3 Σενάριο Πολλαπλών Ροών

3.3.1 Onnxruntime πλαίσιο – χωρίς παρεμβολές

Συγκρίνουμε το QPS που επιτυγχάνουν τα μοντέλα με onnxruntime πλαίσιο υποστήριξης για έναν συνδυασμό διαφορετικών τιμών της μεταβλητής OMP_NUM_THREADS (που παρέχεται από την προδιαγραφή OpenMP API για παράλληλο προγραμματισμό, και ορίζει τον αριθμό των νημάτων ανά «στιγμιότυπο» μιας μηχανής ML inference) και της επιλογής --threads (παρέχεται από τη σουίτα MLPerf Inference benchmark, και αποτελεί τον αριθμό των «στιγμιοτύπων» μιας μηχανής ML inference).

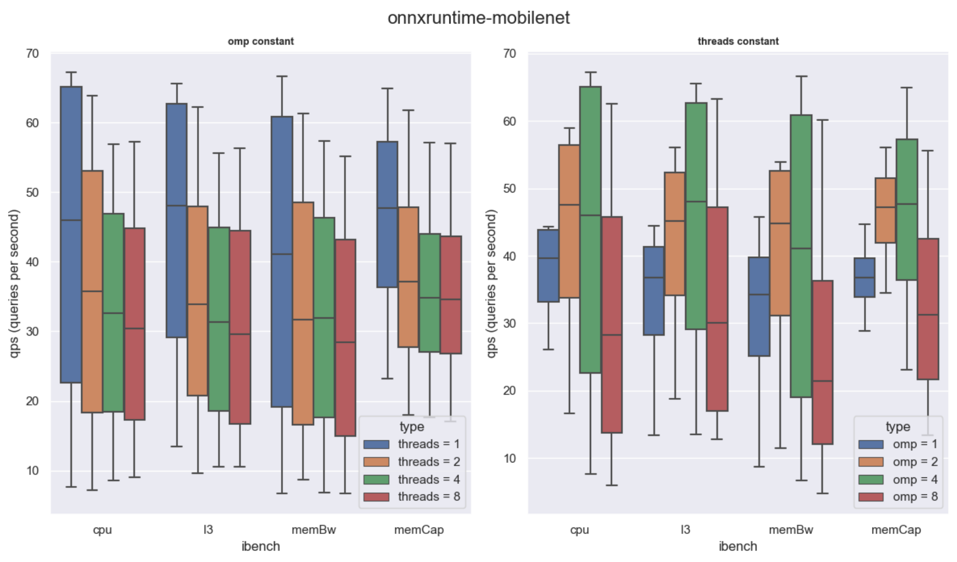


Σχήμα 3.4: Μέτρηση QPS (έγχρωμα τετράγωνα) των MLPerf Inference benchmarks με το πλαίσιο υποστήριξης onnxruntime, για διάφορους συνδυασμούς της μεταβλητής OMP_NUM_THREADS και της επιλογής --threads.

Από το σχήμα, βλέπουμε την ίδια συμπεριφορά και στα τρία μοντέλα μας, να επιτυγχάνουν δηλαδή καλύτερη συνολική απόδοση σε τιμές omp 2 ή περισσότερες, ανεξάρτητα από τον αριθμό των threads. Παρατηρούμε επίσης ότι η καλύτερη απόδοση επιτυγχάνεται όταν η μεταβλητή omp είναι στο 4 και τα threads στο 1 ή το 2 ή όταν έχουμε τη μεταβλητή omp στο 8 και τα threads στο 1.

3.3.2 Onnxruntime πλαίσιο – με παρεμβολές

Κρατώντας σταθερή μία μεταβλητή (είτε τη μεταβλητή OMP_NUM_THREADS είτε την επιλογή --threads) συγκρίνουμε πώς αλλάζει το QPS για τιμές 1, 2, 4 και 8 της άλλης μεταβλητής, σε κάθε κοινόχρηστο πόρο υπό πίεση από 1 εώς 16 πηγές παρεμβολής iBench.

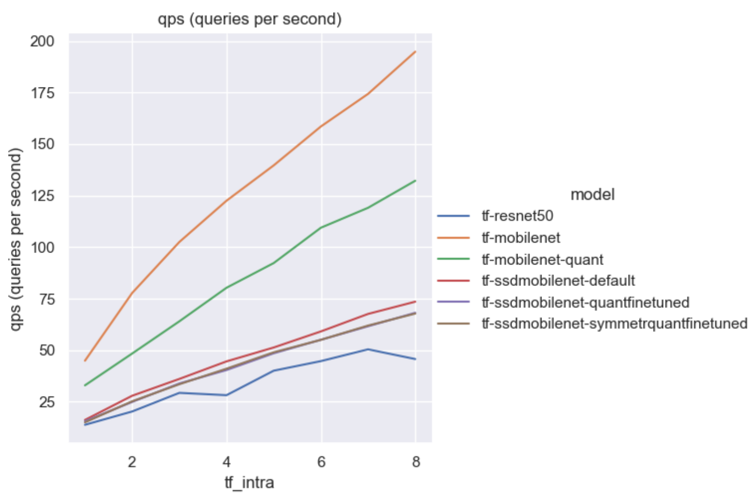


Σχήμα 3.5: Μέτρηση QPS των MLPerf Inference benchmarks με το πλαίσιο υποστήριξης onnxruntime, για διάφορες πηγές παρεμβολών «iBench» σε διαφορετικές εντάσεις. Στο αριστερό σχήμα, η μεταβλητή OMP_NUM_THREADS διατηρείται σταθερή, ενώ στο δεξί σχήμα η επιλογή --threads είναι σταθερή. Ενδεικτικά φαίνεται 1 από τα 3 benchmarks.

Αρχικά, εστιάζουμε στο σχήμα όπου διατηρούμε σταθερή τη μεταβλητή περιβάλλοντος omp (OMP_NUM_THREADS). Παρατηρούμε ότι παίρνουμε σταθερά σημαντικά υψηλότερο QPS για την επιλογή --threads στην τιμή 1 (μπλε πλαίσιο) από ό,τι με υψηλότερες τιμές. Στο σχήμα όπου κρατάμε σταθερή την επιλογή --threads, βλέπουμε την καλύτερη απόδοση σε τιμές Omp 2 (κίτρινο) και 4 (πράσινο), ενώ ενδιαφέρον είναι το γεγονός ότι φαίνεται να έχουν τη χειρότερη απόδοση σε τιμή omp 8 (κόκκινο πλαίσιο).

3.3.3 Tensorflow πλαίσιο – χωρίς παρεμβολές

Μετράμε το QPS (αιτήματα ανά δευτερόλεπτο) που επιτυγχάνουν τα μοντέλα με tensorflow πλαίσιο υποστήριξης για διάφορες τιμές της επιλογής INTRA_OP_PARALLELISM_THREADS που παρέχεται από την πλατφόρμα Tensorflow και ορίζει τον αριθμό των νημάτων που χρησιμοποιούνται σε μια μεμονωμένη πράξη (όπως πολλαπλασιασμός πίνακα και αναγωγές) για παραλληλισμό.

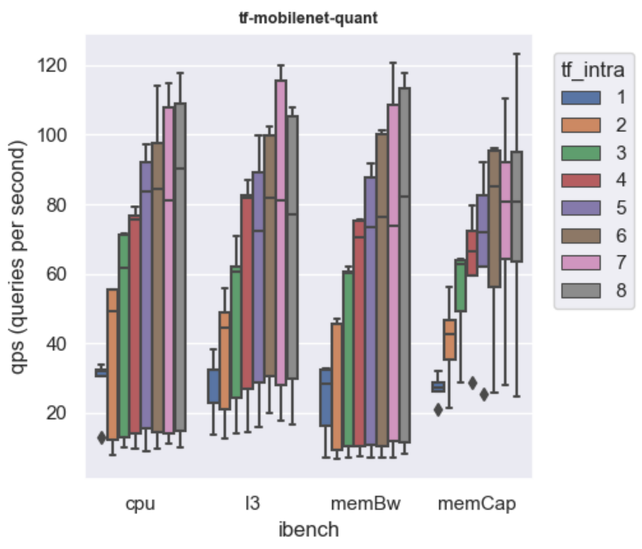


Σχήμα 3.6: Μέτρηση QPS των MLPerf Inference benchmarks με το πλαίσιο υποστήριξης tensorflow, για διάφορες τιμές της επιλογής INTRA_OP_PARALLELISM_THREADS (tf_intra). Τα τρία πρώτα benchmarks, στο υπόμνημα με τα χρώματα στα δεξιά, εκτελούν μια εργασία ταξινόμησης εικόνων, ενώ τα τρία τελευταία εκτελούν μια εργασία ανίχνευσης αντικειμένων.

Εύκολα συμπεραίνουμε από το σχήμα ότι όσο αυξάνεται η τιμή tf_intra, τόσο καλύτερη είναι η απόδοση των μοντέλων. Την μεγαλύτερη κλίση αύξησης για τα μοντέλα που εκτελούν μια εργασία ταξινόμησης εικόνων, την έχει το tf-mobilenet (κίτρινη γραμμή), το οποίο επιτυγχάνει και τα υψηλότερα QPS κάθε στιγμή, με δεύτερο καλύτερο το tf-mobilenet-quant. Για τα μοντέλα που εκτελούν μια εργασία ανίχνευσης αντικειμένων, το μοντέλο με την καλύτερη απόδοση είναι το tf-ssdmobilenet-default (κόκκινη γραμμή), με μικρή διαφορά από τα υπόλοιπα.

3.3.4 Tensorflow πλαίσιο – με παρεμβολές

Για κάθε μοντέλο MLPerf Inference και κάθε κοινόχρηστο πόρο σε πίεση από 1 εώς 16 πηγές παρεμβολής iBench, μετράμε την απόδοση των μοντέλων για διάφορες τιμές της επιλογής INTRA_OP_PARALLELISM_THREADS. Το σχήμα που προκύπτει είναι το ακόλουθο.



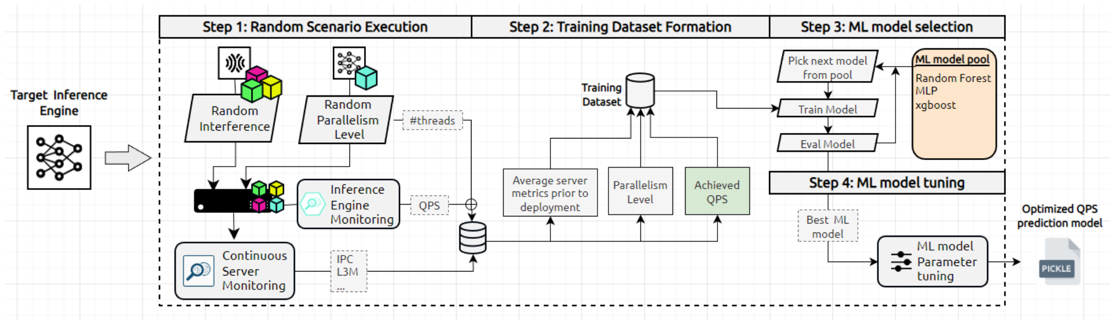
Σχήμα 3.7: Μέτρηση QPS των MLPerf Inference benchmarks με το πλαίσιο υποστήριξης tensorflow, για διάφορες πηγές παρεμβολών «iBench» σε διαφορετικές εντάσεις. Ενδεικτικά φαίνεται 1 από τα 6 benchmarks.

Ξεκινώντας, βλέπουμε μια παρόμοια ανοδική τάση στην απόδοση σε όλα τα MLPerf Inference Benchmarks με το πλαίσιο υποστήριξης tensorflow, καθώς αυξάνεται η τιμή του tf_intra, με τις υψηλότερες τιμές QPS να παρατηρούνται γενικά για τιμές tf_intra 5 και άνω, όπου η διάμεση τιμή της απόδοσης παραμένει σε παρόμοιο υψηλό σημείο. Επίσης παρατηρούμε μικρότερη μεταβλητότητα στην απόδοση, καθώς η ένταση της παρεμβολής ποικίλλει, για όλες τις πηγές παρεμβολής και tf_intra 1. Παρόμοια μικρότερη μεταβλητότητα στην απόδοση έχουμε με την πίεση στην χωρητικότητα της μνήμης για όλες τις τιμές του tf_intra, σε σύγκριση με τις πιέσεις από τις υπόλοιπες πηγές παρεμβολής.

4 Χρονοδρομολογητής

4.1 Εκτός Σύνδεσης: Εκπαίδευση των Regression Μοντέλων Μηχανικής Μάθησης

Για να προσδιορίσουμε ποιο MLPerf Inference benchmark, και με ποια ρύθμιση παραμέτρων, είναι κατάλληλο να εκτελεστεί κάθε φορά, στρεφόμαστε στη βοήθεια ενός συνόλου Regression μοντέλων Μηχανικής Μάθησης (ML) που, κατάλληλα εκπαιδευμένα, θα μπορούν να προβλέψουν την πιθανή απόδοση των benchmarks υπό τις συνθήκες παρεμβολής που υπάρχουν τη στιγμή που στέλνονται για εκτέλεση.


Target Inference Engine
Step 1: Random Scenario Execution
Step 2: Training Dataset Formation
Step 3: ML model selection
Random Interference
Random Parallelism Level
#threads
Inference Engine Monitoring
QPS
Continuous Server Monitoring
IPC
L3M
...
Training Dataset
Average server metrics prior to deployment
Parallelism Level
Achieved QPS
Pick next model from pool
Train Model
Eval Model
ML model pool
Random Forest
MLP
xgboost
Step 4: ML model tuning
Best ML model
ML model Parameter tuning
Optimized QPS prediction model
PICKLE

Σχήμα 4.1: Διάγραμμα χρονοδρομολογητή εκτός σύνδεσης της (1) εκτέλεσης σεναρίων εκπαίδευσης, (2) του σχηματισμού συνόλου δεδομένων εκπαίδευσης, (3) της εκπαίδευσης και αξιολόγησης των μοντέλων ML Regression, (4) της επιλογής και βελτιστοποίησης ενός ML μοντέλου, για κάθε MLPerf Inference benchmark

Η εκπαίδευση των Regression μοντέλων Μηχανικής Μάθησης έγινε εκ των προτέρων, συλλέγοντας ένα σύνολο από διάφορες μετρήσεις συστήματος (σχετικές με τους πυρήνες της ΚΜΕ, την κρυφή μνήμη και την κύρια μνήμη) σε διαφορετικά σενάρια παρεμβολών και μετρώντας την απόδοση των MLPerf Inference benchmarks. Σε κάθε σενάριο παρεμβολής, τρέχουμε ένα τυχαίο αριθμό από iBench πηγές παρεμβολής που πιέζουν ένα τυχαίο πόρο του συστήματος το καθένα, για όλη τη διάρκεια του σεναρίου. Έπειτα συλλέγουμε τις μετρήσεις συστήματος για 20 δευτερόλεπτα και κρατάμε τον μέσο όρο τους, και τρέχουμε ένα τυχαίο benchmark με τυχαίο επίπεδο παραλληλισμού (που καθορίζεται από τις τιμές, που κυμαίνονται από 1 εώς 8, των OMP_NUM_THREADS και --threads για τα benchmarks με onnxruntime πλαίσιο υποστήριξης, και του INTRA_OP_PARALLELISM_THREADS για τα benchmarks με tensorflow πλαίσιο υποστήριξης) σε σενάριο πολλαπλών ροών για 60 δευτερόλεπτα. Οι μετρήσεις του συστήματος, η τιμή του επιπέδου παραλληλισμού και το QPS που πετυχαίνει το benchmark σε αυτές τις συνθήκες αποτελούν μια γραμμή του συνόλου δεδομένων της εκπαίδευσης.

Με το σύνολο δεδομένων εκπαίδευσης που προέκυψε, εκπαιδεύσαμε διάφορα μοντέλα ML Regression και τα αξιολογήσαμε με βάση τη βαθμολογία διασταυρούμενης επικύρωσης που πετύχαιναν.

MLPerf Inference Benchmarks (image classification task)					
Algorithm	**onnx-resnet50**	**tf-resnet50**	**onnx-mobilenet**	**tf-mobilenet**	**tf-mobilenet-quant**
Linear	-1.3e+12	-5.4e+23	-4.8e+11	-1.2e+23	0.813
Ridge	-6.1e+11	-1.6e+12	-1.6e+11	-2.9e+12	0.814
Lasso	0.679	0.745	0.801	0.783	0.797
Elastic Net	0.679	0.744	0.803	0.783	0.798
Bayesian Ridge	-3e+9	-4.2e+10	-1.7e+11	-9.2e+11	0.812
SGD	-5.1e+43	-6.4e+43	-1.8e+43	-5.1e+42	-5.1e+42
SVR	0.253	0.201	0.295	0.232	0.202
k-NN	0.236	0.19	0.324	0.239	0.266
Gaussian Process	-3	-1.9	-3.5	-2.1	-1.6
Decision Tree	0.632	0.921	0.835	0.93	0.92

Random Forest	0.825	0.955	0.909	0.962	0.959
MLP	-3e+8	-4e+8	-1.2e+8	-3.6e+7	-5.6e+6
XGBoost	0.843	0.962	0.914	0.962	0.956

Πίνακας 4.1: Βαθμολογία κατά 10-πλη διασταυρούμενη επικύρωση διαφορετικών Μοντέλων ML Regression για την πρόβλεψη του QPS διαφόρων MLPerf Inference benchmarks που εκτελούν μια εργασία ταξινόμησης εικόνας.

	MLPerf Inference Benchmarks (object detection task)			
Algorithm	**onnx-ssd-mobilenet**	**tf-ssd-mobilenet**	**tf-ssd-mobilenet-quant-finetuned**	**tf-mobilenet-symmetr-quant-finetuned**
Linear	-1.7e+13	0.731	-1.1e+12	-6.4e+9
Ridge	-8.9e+12	0.767	-6.7e+11	-8.4e+10
Lasso	0.838	0.792	0.763	0.769
Elastic Net	0.842	0.791	0.762	0.765
Bayesian Ridge	-1.3e+13	0.797	-6.6e+11	-6.7e+10
SGD	-1.3e+45	-2e+43	-2.6e+43	-2e+43
SVR	0.332	0.132	0.139	0.772
k-NN	0.327	0.134	0.126	0.55
Gaussian Process	-2.4	-2	-2.2	-2.1
Decision Tree	0.878	0.927	0.926	0.929
Random Forest	0.942	0.963	0.954	0.955
MLP	-1.4e+10	-1.3e+5	-8.4e+8	-2.3e+6
XGBoost	0.941	0.965	0.956	0.961

Πίνακας 4.2: Βαθμολογία κατά 10-πλη διασταυρούμενη επικύρωση διαφορετικών Μοντέλων ML Regression για την πρόβλεψη του QPS διαφόρων MLPerf Inference benchmarks που εκτελούν μια εργασία αναγνώρισης αντικειμένων.

Στη συνέχεια επιλέξαμε το πιο ακριβές μοντέλο Regression ML, με την μεγαλύτερη βαθμολογία στην πρόβλεψη του QPS των benchmarks, για χρήση στον χρονοδρομολογητή μας. Δεδομένου ότι το XGBoost πετυχαίνει την καλύτερη βαθμολογία στα περισσότερα benchmarks, και το Random forest ξεπερνά το XGBoost μόνο σε 2 από τις συνολικά 9 προβλέψεις, και μόνο λίγο, για λόγους απλότητας, θα χρησιμοποιήσουμε το XGBoost αλγόριθμο Regression ML για την πρόβλεψη του QPS για όλα τα MLPerf Inference benchmarks.

Τέλος, ρυθμίσαμε τις υπερπαραμέτρους του XGBoost, για κάθε benchmark, για να βελτιστοποιήσουμε περαιτέρω την ακρίβειά του.

MLPerf Inference benchmarks	**XGBoost Regression (default parameters)**	**XGBoost Regression (tuned parameters)**
onnxruntime-resnet50	0.843	0.877
tensorflow-resnet50	0.962	0.964
onnxruntime-mobilenet	0.914	0.924
tensorflow-mobilenet	0.962	0.97
tensorflow-mobilenet-quantized	0.956	0.966
onnxruntime-ssd-mobilenet	0.941	0.943

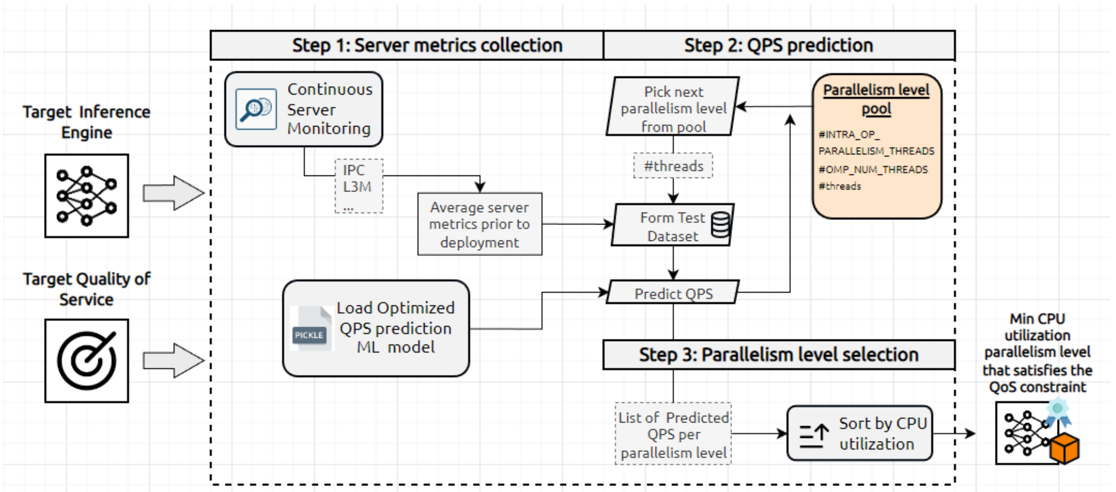
tensorflow-ssd-mobilenet	0.965	0.971
tensorflow-ssd-mobilenet-quantized-finetuned	0.956	0.964
tensorflow-ssd-mobilenet-symmetrically-quantized-finetuned	0.961	0.966

Πίνακας 4.3: Σύγκριση βαθμολογίας κατά 10-πλη διασταυρούμενη επικύρωση του αλγόριθμου XGBoost για ML Regression πριν και μετά τη βελτιστοποίηση των υπερπαραμέτρων

4.2 Σε Σύνδεση: Προγνωστικός Χρονοδρομολογητής Inference Μηχανών με Επίγνωση των Παρεμβολών και των Πόρων

4.2.1 Ανάπτυξη για Συγκεκριμένο Μοντέλο

Ο μηχανισμός χρονοδρομολόγησης Inference μηχανών με συγκεκριμένο μοντέλο που αναπτύξαμε λαμβάνει υπ' όψη το τρέχον επίπεδο συνθηκών παρεμβολής στο σύστημα πριν από την επιλογή του MLPerf Inference benchmark για εκτέλεση στο σύμπλεγμα σε ένα σενάριο πολλαπλών ροών. Επίσης, στοχεύει να στείλει το benchmark με μια διαμόρφωση (επίπεδο παραλληλισμού) που του επιτρέπει να κάνει όσο το δυνατόν λιγότερη χρήση πόρων του συστήματος, ενώ καταφέρνει να ικανοποιήσει έναν απαιτούμενο περιορισμό QoS.



Step 1: Server metrics collection
Step 2: QPS prediction
Target Inference Engine
Continuous Server Monitoring
IPC
L3M
...
Average server metrics prior to deployment
Pick next parallelism level from pool
#threads
Form Test Dataset
Parallelism level pool
#INTRA_OP_PARALLELISM_THREADS
#OMP_NUM_THREADS
#threads
Target Quality of Service
Load Optimized QPS prediction ML model
PICKLE
Predict QPS
Step 3: Parallelism level selection
List of Predicted QPS per parallelism level
Sort by CPU utilization
Min CPU utilization parallelism level that satisfies the QoS constraint

Σχήμα 4.2: Σε σύνδεση, διάγραμμα χρονοδρομολογητή για συγκεκριμένο μοντέλο, της διαδικασίας επιλογής επιπέδου παραλληλισμού μιας inference μηχανής, προκειμένου να ικανοποιηθεί ένας QoS στόχος με ελάχιστη χρήση πόρων.

Ο χρονοδρομολογητής για συγκεκριμένο μοντέλο παίρνει ως είσοδο το όνομα του MLPerf Inference benchmark που πρόκειται να εκτελέσει είτε μια ταξινόμηση εικόνας είτε μια εργασία ανίχνευσης αντικειμένων. Λαμβάνει επίσης το επιθυμητό QPS που στοχεύει να έχει ως κατώτερο όριο για την απόδοση του benchmark.

Όταν ένα αίτημα χρονοδρομολόγησης έρχεται στον χρονοδρομολογητή μας, αυτός συλλέγει αμέσως ένα σύνολο μετρήσεων συστήματος, σχετικά με τα τρέχοντα επίπεδα χρήσης της CPU και της μνήμης στο σύμπλεγμα, κάθε δευτερόλεπτο, για μια καθορισμένη χρονική διάρκεια, και διατηρεί τη μέση τιμή κάθε μετρικής κατά τη διάρκεια αυτής της χρονικής περιόδου. Τότε, φορτώνει το αποθηκευμένο μοντέλο XGBoost Regression που έχει εκπαιδευτεί για το MLPerf Inference benchmark που πρόκειται να δρομολογήσει. Ανάλογα με το πλαίσιο υποστήριξης (tensorflow ή onnxruntime) που θα χρησιμοποιήσει το benchmark, ορίζει μια τιμή στις αντίστοιχες μεταβλητές (INTRA_OP_PARALLELISM_THREADS για το tensorflow πλαίσιο, OMP_NUM_THREADS και --threads επιλογή για το onnxruntime πλαίσιο) του benchmark. Ο χρονοδρομολογητής ζητά από το Regression μοντέλο να προβλέψει το QPS που θα επιτύχει το benchmark, εάν εκτελεστεί με αυτό το επίπεδο παραλληλισμού, υπό τις συνθήκες παρεμβολής που υποδεικνύονται από τις μετρήσεις του συστήματος που συλλέγονται. Επαναλαμβάνει αυτή τη διαδικασία για όλους τους πιθανούς συνδυασμούς τιμών των μεταβλητών του benchmark (κάθε τιμή κυμαίνεται από 1 έως 8), αποθηκεύοντας όλες τις προβλέψεις QPS του Regression μοντέλου (μία πρόβλεψη QPS για κάθε μοναδική διαμόρφωση του benchmark, κάτω από το ίδια τρέχουσα κατάσταση συστήματος), καθώς και τις τιμές των μεταβλητών που οδήγησαν σε αυτήν την πρόβλεψη, σε έναν πίνακα. Ο χρονοδρομολογητής προχωρά στην ταξινόμηση του πίνακα κατά την ποσότητα χρήσης της CPU που έχει κάθε καταχώρηση, με αύξουσα σειρά, και επιλέγει το επίπεδο παραλληλισμού, με το οποίο θα δρομολογήσει το benchmark, που θα οδηγήσει ενδεχομένως σε ένα QPS που θα ικανοποιεί τον περιορισμό QoS (με βάση την τιμή του προβλεπόμενου QPS για κάθε επίπεδο παραλληλισμού στον πίνακα) με τη μικρότερη δυνατή χρήση της CPU.

4.2.2 Προσέγγιση Χωρίς Μοντέλο

Στην προσέγγιση χωρίς μοντέλο στον μηχανισμό χρονοδρομολόγησης των inference μηχανών, εισάγουμε μια νέα διεπαφή χωρίς μοντέλο, όπου οι προγραμματιστές καθορίζουν μόνο την εργασία που θέλουν να εκτελέσουν (ταξινόμηση εικόνας, ανίχνευση αντικειμένων) και την απόδοση υψηλού επιπέδου που απαιτούν ως στόχο QoS. Ο χρονοδρομολογητής χωρίς μοντέλο, λοιπόν, επιλέγει την καλύτερη inference μηχανή, από μια ομάδα εγγεγραμμένων, εκπαιδευμένων inference μοντέλων για συγκεκριμένη εργασία, με το κατάλληλο επίπεδο παραλληλισμού για να ολοκληρώσει αυτή τη δουλειά με τον λιγότερο απαιτητικό σε πόρους τρόπο, ενώ στοχεύει να ικανοποιήσει τον περιορισμό QoS υπό τις τρέχουσες συνθήκες παρεμβολών στο σύστημα.

Ο χρονοδρομολογητής χωρίς μοντέλο λαμβάνει ως είσοδο το όνομα της εργασίας που πρόκειται να εκτελέσουν τα MLPerf Inference benchmarks, η οποία είναι είτε ταξινόμηση εικόνας είτε ανίχνευση αντικειμένων. Λαμβάνει επίσης ένα στόχο QoS, ως κατώτερο όριο για την απόδοση των benchmarks.

Όταν ένα αίτημα χρονοδρομολόγησης έρχεται στον χρονοδρομολογητή χωρίς μοντέλο, όπως κάνει αρχικά και ο χρονοδρομολογητής για συγκεκριμένο μοντέλο, συλλέγει αμέσως ένα σύνολο μετρήσεων συστήματος, σχετικά με τα τρέχοντα επίπεδα χρήσης της CPU και της μνήμης στο σύμπλεγμα, κάθε δευτερόλεπτο, για μια καθορισμένη χρονική διάρκεια, και διατηρεί τη μέση τιμή κάθε μετρικής κατά τη διάρκεια αυτής της χρονικής περιόδου. Έπειτα, πλοηγείται στο χώρο των καταχωρημένων inference μηχανών για την ζητούμενη εργασία. Για κάθε inference μηχανή εκεί, προχωρά με παρόμοιο τρόπο με τον χρονοδρομολογητή με συγκεκριμένο μοντέλο, έως ότου έχει το επίπεδο παραλληλισμού και την αντίστοιχη πρόβλεψη QPS, κάθε inference μηχανής, που πρόκειται να ικανοποιήσει τον περιορισμό QoS με την ελάχιστη χρήση CPU, υπό τις τρέχουσες συνθήκες παρεμβολών. Στη συνέχεια, ο χρονοδρομολογητής χωρίς μοντέλο αποθηκεύει όλες τις τριπλέτες {inference μηχανή – επιλεγμένο επίπεδο παραλληλισμού – προβλεπόμενο QPS σε αυτό το επίπεδο παραλληλισμού} σε μια λίστα και την ταξινομεί με βάση το ποσοστό χρήσης της CPU που έχει κάθε καταχώρηση, με αύξουσα σειρά. Τέλος, επιλέγει να δρομολογήσει την inference μηχανή, στο επίπεδο παραλληλισμού με την οποία συνδυάστηκε στην τριπλέτα, που θα επιτύχει δυνητικά ένα QPS που ικανοποιεί τον περιορισμό QoS με τη μικρότερη δυνατή χρήση CPU.

5 Αξιολόγηση

5.1 Περιγραφή Σεναρίων

Προκειμένου να αξιολογήσουμε τον χρονοδρομολογητή inference μηχανών με συγκεκριμένο μοντέλο ή χωρίς μοντέλο, δημιουργήσαμε 3 διαφορετικά σενάρια παρεμβολών για να ασκήσουμε πίεση κυμαινόμενης έντασης στους κοινόχρηστους πόρους του συστήματος ενώ εκτελούμε τα πειράματά μας.

Σε κάθε σενάριο παρεμβολής, τρέχουμε ένα νέο σύνολο από 0 εώς 2 πηγών παρεμβολής iBench κάθε 10 με 30 δευτερόλεπτα, καθένα από τα οποία ασκεί πίεση σε έναν τυχαία επιλεγμένο κοινόχρηστο πόρο για ένα τυχαίο χρονικό διάστημα (70s έως 220s). Με αυτό τον τρόπο φτιάξαμε 3 σενάρια παρεμβολών.

5.2 Περιγραφή Πειραμάτων

Σε κάθε πείραμα, ξεκινάμε ένα σενάριο παρεμβολής και, παράλληλα, εκτελούμε τα MLPerf Inference benchmarks, χρησιμοποιώντας τον χρονοδρομολογητή μας. Του ζητάμε επίσης η απόδοση κάθε benchmark να ικανοποιεί έναν συγκεκριμένο περιορισμό QoS. Τα benchmarks εκτελούνται σε σενάριο πολλαπλών ροών, επομένως αποκλείουμε το tflite-mobilenet από τα πειράματά μας, καθώς δεν λειτουργεί σε αυτό το σενάριο. Κάθε benchmark δρομολογείται 10 συνεχόμενες φορές από τον ίδιο χρονοδρομολογητή και εκτελείται για 30 δευτερολέπτα στο σύμπλεγμα κάθε φορά. Καθ' όλη τη διάρκεια του πειράματος, η ένταση της παρεμβολής στο σύμπλεγμα αλλάζει σύμφωνα με το επιλεγμένο σενάριο παρεμβολής. Ο χρονοδρομολογητής μας συλλέγει τις μετρήσεις του συστήματος για 5 δευτερόλεπτα, πριν από την δρομολόγηση του επόμενου benchmark κάθε φορά.

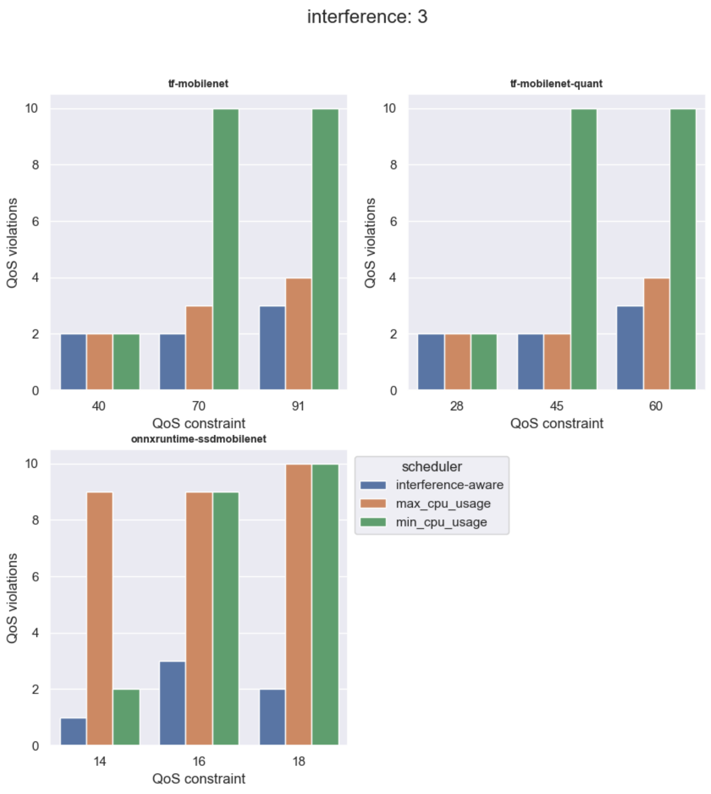
Επαναλαμβάνουμε κάθε πείραμα για 3 διαφορετικούς περιορισμούς QoS και με τη χρήση 3 διαφορετικών χρονοδρομολογητών: τον χρονοδρομολογητή μας με επίγνωση παρεμβολών και πόρων, έναν χρονοδρομολογητή ελάχιστης χρήσης CPU και έναν χρονοδρομολογητή μέγιστης χρήσης CPU. Αξιολογούμε επίσης τους χρονοδρομολογητές σε 3 διαφορετικά σενάρια παρεμβολών και συγκρίνουμε τα αποτελέσματά τους.

Στα πειράματα με τον χρονοδρομολογητή χωρίς μοντέλο, έχουμε τις παρακάτω διαφοροποιήσεις. Του ζητάμε να εκτελέσει μια εργασία (ταξινόμηση εικόνας ή ανίχνευση αντικειμένων), ενώ κάθε benchmark που επιλέγει δρομολογείται 20 συνεχόμενες φορές από τον ίδιο χρονοδρομολογητή και εκτελείται για 30 δευτερολέπτα στο σύμπλεγμα κάθε φορά.

Επαναλαμβάνουμε κάθε πείραμα για 3 διαφορετικούς περιορισμούς QoS, και συγκρίνουμε τα αποτελέσματα του χρονοδρομολογητή χωρίς μοντέλο και του χρονοδρομολογητή με συγκεκριμένο μοντέλο σε ένα σενάριο παρεμβολών.

5.3 Αποτελέσματα Χρονοδρομολογητή Inference Μηχανών Με Συγκεκριμένο Μοντέλο & Σύγκριση Χρονοδρομολογητών

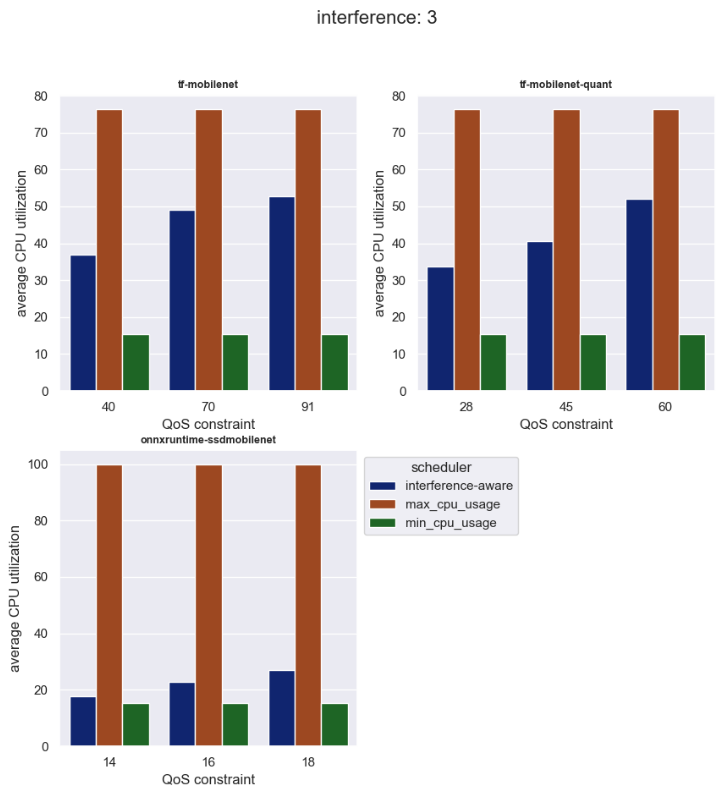
5.3.1 Παραβιάσεις του περιορισμού QoS



Σχήμα 5.1: Σύγκριση αριθμού παραβιάσεων QoS μεταξύ του χρονοδρομολογητή με επίγνωση παρεμβολών, του χρονοδρομολογητή μέγιστης χρήσης CPU και του χρονοδρομολογητή ελάχιστης χρήσης CPU, για 3 διαφορετικούς περιορισμούς QoS. Ενδεικτικά φαίνονται 3 από τα 9 benchmarks, σε 1 από τα 3 σενάρια παρεμβολών.

Παρατηρούμε ότι ο χρονοδρομολογητής μας καταφέρνει να διατηρεί τον αριθμό των παραβιάσεων QoS σημαντικά μικρότερο από ό,τι ο χρονοδρομολογητής ελάχιστης χρήσης CPU σε όλες τις περιπτώσεις σε περιορισμό μεσαίου ή υψηλού QoS και αποδίδει καλύτερα ή ίσα με τον χρονοδρομολογητή ελάχιστης χρήσης CPU σε όλες σχεδόν τις περιπτώσεις σε χαμηλό QoS περιορισμό. Επιπλέον, πετυχαίνουμε αξιοσημείωτα καλύτερη απόδοση από τον χρονοδρομολογητή μέγιστης χρήσης CPU, στα benchmarks με πλαίσιο υποστήριξης onnxruntime. Αυτό συμβαίνει διότι σε αυτό το πλαίσιο υποστήριξης, τα benchmarks έχουν την μέγιστη απόδοση σε ένα μέτριο επίπεδο παραλληλισμού, ενώ έχουν πτώση απόδοσης σε υψηλό επίπεδο παραλληλισμού. Στα tensorflow πλαίσια υποστήριξης, ο αριθμός παραβιάσεων QoS του χρονοδρομολογητή μας είναι παρόμοιος με αυτόν του χρονοδρομολογητή μέγιστης χρήσης CPU, αν και ο δικός μας το πετυχαίνει χωρίς να χρησιμοποιεί συνεχώς τους πόρους του συμπλέγματος στη μέγιστη χωρητικότητα.

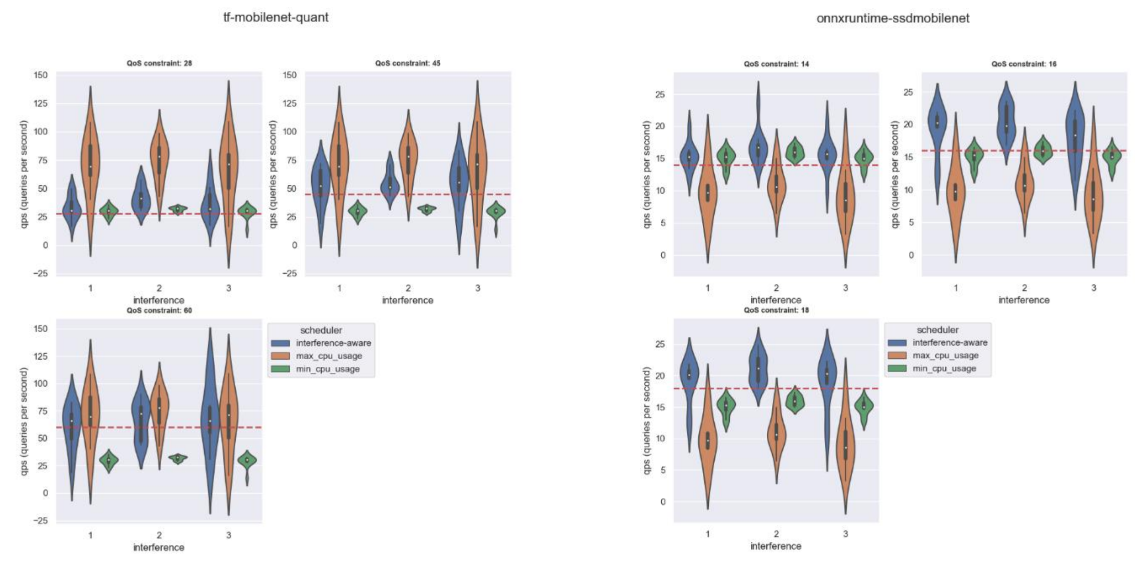
5.3.2 Αξιοποίηση πόρων



Σχήμα 5.2: Σύγκριση μέσου ποσοστού χρήσης CPU μεταξύ του χρονοδρομολογητή με επίγνωση παρεμβολών, του χρονοδρομολογητή μέγιστης χρήσης CPU και του χρονοδρομολογητή ελάχιστης χρήσης CPU, για 3 διαφορετικούς περιορισμούς QoS. Ενδεικτικά φαίνονται 3 από τα 9 benchmarks, σε 1 από τα 3 σενάρια παρεμβολών.

Παρατηρούμε ότι ο χρονοδρομολογητής μας καταφέρνει να επιτύχει μια μέση χρήση της CPU κάπου ανάμεσα στην ελάχιστη χρήση και στη μέγιστη χρήση της CPU, ανάλογα τον περιορισμό QoS και τις παρεμβολές στους πόρους του συστήματος. Πλησιάζει ή ξεπερνάει τον περιορισμό QoS για την πλειονότητα των δρομολογήσεων των benchmarks, σε αντίθεση με τον χρονοδρομολογητή ελάχιστης χρήσης CPU, ενώ επίσης το κάνει με μικρότερη χρήση CPU από τον χρονοδρομολογητή μέγιστης χρήσης CPU.

5.3.3 Κατανομή απόδοσης

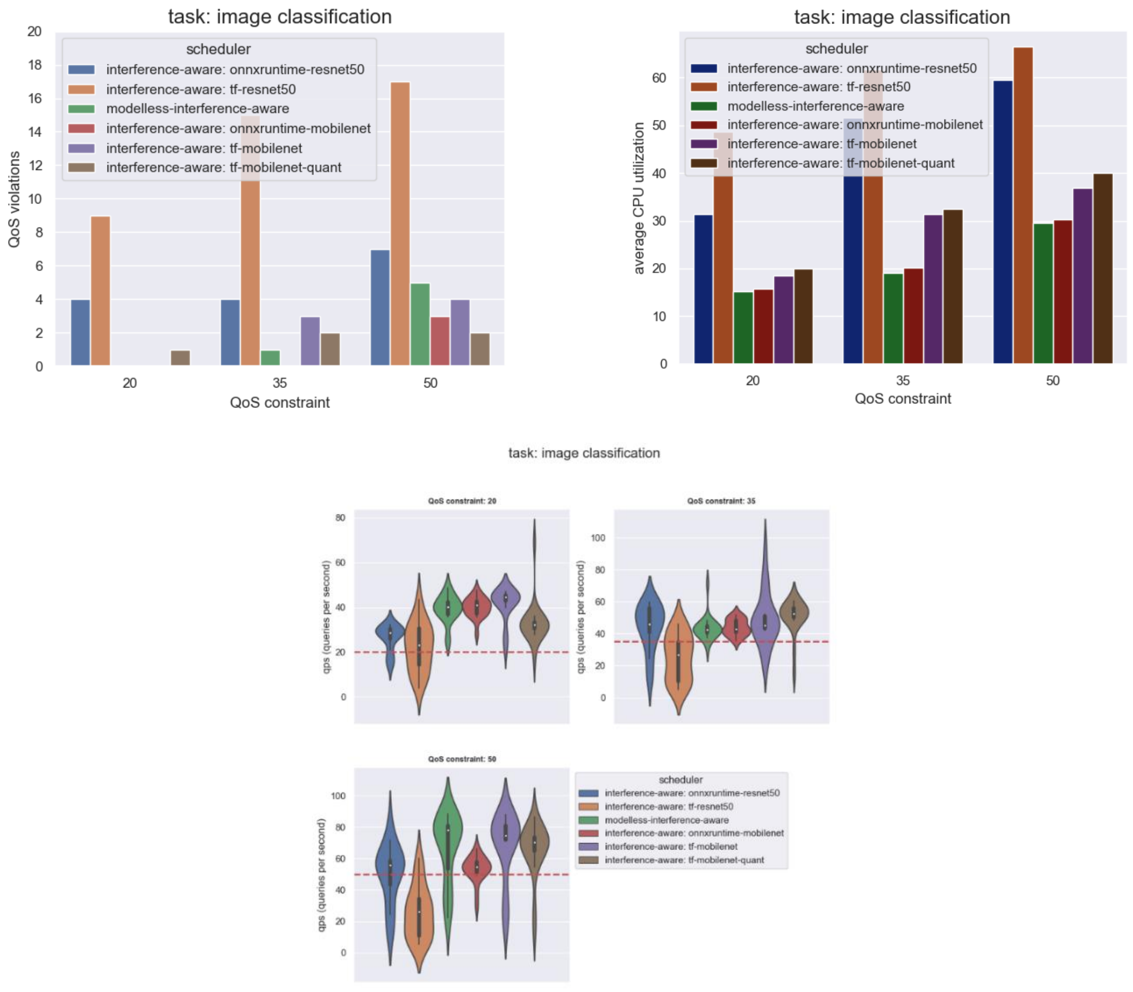


Σχήμα 5.3: Σύγκριση κατανομής QPS μεταξύ του του χρονοδρομολογητή με επίγνωση παρεμβολών, του χρονοδρομολογητή μέγιστης χρήσης CPU και του χρονοδρομολογητή ελάχιστης χρήσης CPU, για 3 διαφορετικούς περιορισμούς QoS. Ενδεικτικά φαίνονται 2 από τα 9 benchmarks, σε 3 σενάρια παρεμβολών.

Από τα σχήματα βλέπουμε ότι ο χρονοδρομολογητής μας καταφέρνει να έχει μικρότερη μεταβλητότητα στην απόδοση σε σχέση με τον χρονοδρομολογητή μέγιστης χρήσης CPU, ιδιαίτερα σε μικρό και μεσαίο περιορισμό QoS. Επίσης, ο χρονοδρομολογητής μας παρουσιάζει μια κατανομή απόδοσης με την υψηλότερη πιθανότητα τα QPS να βρίσκονται πάνω από το στόχο QoS, για όλους τους περιορισμούς QoS. Έτσι καταφέρνει να έχει περισσότερες εγγυήσεις του QoS σε σχέση με τον χρονοδρομολογητή ελάχιστης χρήσης CPU, ο οποίος καταφέρνει να πετύχει αξιόπιστα τον στόχο QoS μόνο με χαμηλό περιορισμό QoS, ενώ η απόδοση του δεν μπορεί να φτάσει το στόχο QoS εάν είναι σε μεσαία τιμή ή υψηλότερη.

5.4 Αποτελέσματα Χρονοδρομολογητή Inference Μηχανών Χωρίς Μοντέλο & Σύγκριση Χρονοδρομολογητών

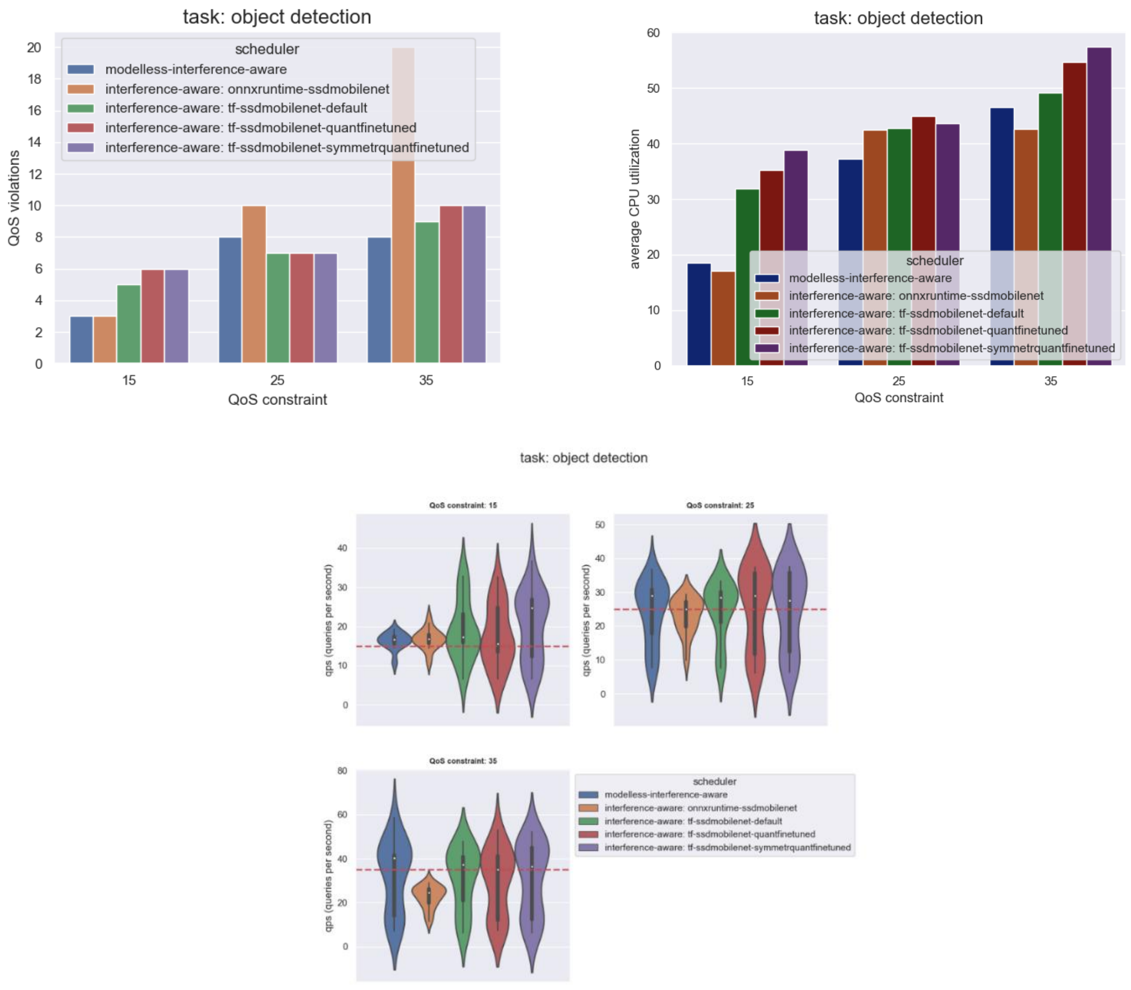
5.4.1 Εργασία Ταξινόμησης Εικόνων



Σχήμα 5.4: Σύγκριση (1) του αριθμού παραβιάσεων QoS, (2) του μέσου ποσοστού χρήσης CPU, (3) της κατανομής QPS μεταξύ του χρονοδρομολογητή με επίγνωση παρεμβολών με συγκεκριμένο μοντέλο και του χρονοδρομολογητή με επίγνωση παρεμβολών χωρίς μοντέλο, σε 3 διαφορετικούς περιορισμούς QoS για την ML εργασία ταξινόμησης εικόνων.

Ξεκινώντας την ανάλυση με την εργασία ταξινόμησης εικόνων, σε χαμηλό και μεσαίο στόχο QoS, ο χρονοδρομολογητής χωρίς μοντέλο χρησιμοποιεί κυρίως το benchmark onnxruntime-mobilenet, το οποίο δεν παραβιάζει τον περιορισμό. Στον υψηλό περιορισμό QoS, χρησιμοποιεί το benchmark tf-mobilenet τις περισσότερες φορές, το οποίο έχει την ικανότητα να πετυχαίνει υψηλά QPS με μικρή χρήση CPU. Παρατηρούμε ότι καταφέρνει να διατηρήσει τη μέση χρήση της CPU, στο ελάχιστο, σε όλους τους περιορισμούς QoS. Τέλος, ο χρονοδρομολογητής χωρίς μοντέλο έχει παρόμοια μεταβλητότητα στην απόδοση με το benchmark που επιλέγει να χρησιμοποιεί περισσότερο σε κάθε περίπτωση διαφορετικού QoS. Έτσι καταφέρνει να κρατάει τη διάμεση τιμή του QPS πάνω από τους περιορισμούς QoS.

5.4.2 Εργασία Ανίχνευσης Αντικειμένων



Σχήμα 5.5: Σύγκριση (1) του αριθμού παραβιάσεων QoS, (2) του μέσου ποσοστού χρήσης CPU, (3) της κατανομής QPS μεταξύ του χρονοδρομολογητή με επίγνωση παρεμβολών με συγκεκριμένο μοντέλο και του χρονοδρομολογητή με επίγνωση παρεμβολών χωρίς μοντέλο, σε 3 διαφορετικούς περιορισμούς QoS για την ML εργασία ανίχνευσης αντικειμένων.

Όσον αφορά την εργασία ανίχνευσης αντικειμένων, ο χρονοδρομολογητής χωρίς μοντέλο χρησιμοποιεί ως επί το πλείστον το benchmark onnxruntime-ssdmobilenet σε χαμηλό περιορισμό QoS, με αποτέλεσμα παρόμοια ικανοποιητική απόδοση με τον χρονοδρομολογητή με συγκεκριμένο μοντέλο που χρησιμοποιεί μόνο το onnxruntime-ssdmobilenet, που πληροί το στόχο QoS σχεδόν κάθε φορά. Για τους μεσαίους και υψηλούς περιορισμούς QoS, ο χρονοδρομολογητής χωρίς μοντέλο στρέφεται στο benchmark tf-ssdmobilenet για τις περισσότερες από τις δρομολογήσεις. Ως αποτέλεσμα, έχει περίπου τις ίδιες παραβιάσεις QoS, κατά μέσο όρο, με τον χρονοδρομολογητή με συγκεκριμένο μοντέλο που δρομολογεί το tf-ssdmobilenet, και αποδίδει σημαντικά καλύτερα από τον χρονοδρομολογητή με το benchmark onnxruntime-ssdmobilenet σε αυτά τα QoS. Επίσης, παρατηρούμε ότι καταφέρνει να διατηρήσει τη μέση χρήση της CPU, στο ελάχιστο, σε όλους τους περιορισμούς QoS. Τέλος, ο χρονοδρομολογητής χωρίς μοντέλο έχει παρόμοια μεταβλητότητα στην απόδοση με το benchmark που επιλέγει να χρησιμοποιεί περισσότερο σε κάθε περίπτωση διαφορετικού QoS. Έτσι καταφέρνει να κρατάει τη διάμεση τιμή του QPS πάνω από τους περιορισμούς QoS.

6 Σύνοψη

Σε αυτή τη διπλωματική εργασία, σχεδιάσαμε ένα προγνωστικό πλαίσιο χρονοδρομολόγησης με επίγνωση των παρεμβολών και πόρων, για ML inference μηχανές, προκειμένου να ανταποκρίνεται στους περιορισμούς QoS των εφαρμογών, και παράλληλα να είναι όσο το δυνατόν λιγότερο απαιτητικό σε πόρους συστήματος. Επεκτείναμε επίσης τη σχεδίασή μας σε ένα πλαίσιο χρονοδρομολόγησης χωρίς μοντέλο για να αντιμετωπίσουμε τον μεγάλο αριθμό διαφορετικών inference μηχανών. Αξιολογήσαμε το προτεινόμενο πλαίσιο χρονοδρομολόγησης μας, χρησιμοποιώντας ένα σύνολο inference μηχανών από τη σουίτα MLPerf Inference Benchmark στο σύστημα Kubernetes, χρησιμοποιώντας διαφορετικά σενάρια παρεμβολών, που δημιουργήθηκαν με τη χρήση της σουίτας φόρτου εργασίας iBench. Δείξαμε ότι για ένα σύνολο διαφορετικών απαιτούμενων περιορισμών QoS, το πλαίσιο χρονοδρομολόγησης μας παραβιάζει τους περιορισμούς QoS, κατά μέσο όρο, λιγότερο συχνά, με μικρή μεταβλητότητα απόδοσης γύρω από το στόχο QoS και με μέτρια χρήση των CPU πόρων ανάλογα με το QoS στόχο και τον φόρτο εργασίας στους πόρους, σε σύγκριση με το σύστημα παροχής ML inference ελάχιστης και μέγιστης χρήσης CPU. Τέλος, δείξαμε ότι η προσέγγισή μας χωρίς μοντέλο βελτιώνει περαιτέρω τον μέσο αριθμό παραβιάσεων των περιορισμών QoS και τη μέση χρήση της CPU του πλαισίου χρονοδρομολόγησης μας.

Chapter 1

Introduction

The number of applications relying on inference from Machine Learning (ML) models is already large and expected to keep growing. Facebook, for instance, serves tens-of-trillions of inference queries per day [1]. These inference requests are usually being offloaded to the Cloud. The increasing effectiveness of Machine Learning (ML) and the advent of Cloud services is producing rapid growth of Machine Learning as a Service (MLaaS) platforms such as IBM Watson, Google Cloud Vertex AI, Amazon ML and Elastic Inference, and Microsoft Azure ML, where the Cloud providers offer their inference serving systems [2]. As an analysis, from Google Trends, of 10 years of search interest trends in data science related topics shows below, interest in "Machine Learning" has been slowly getting traction, when in 2017 it surpassed the most popular trend at the time "Big Data", and has been on a rapid rise ever since, being nowadays the most popular search term by a large margin.

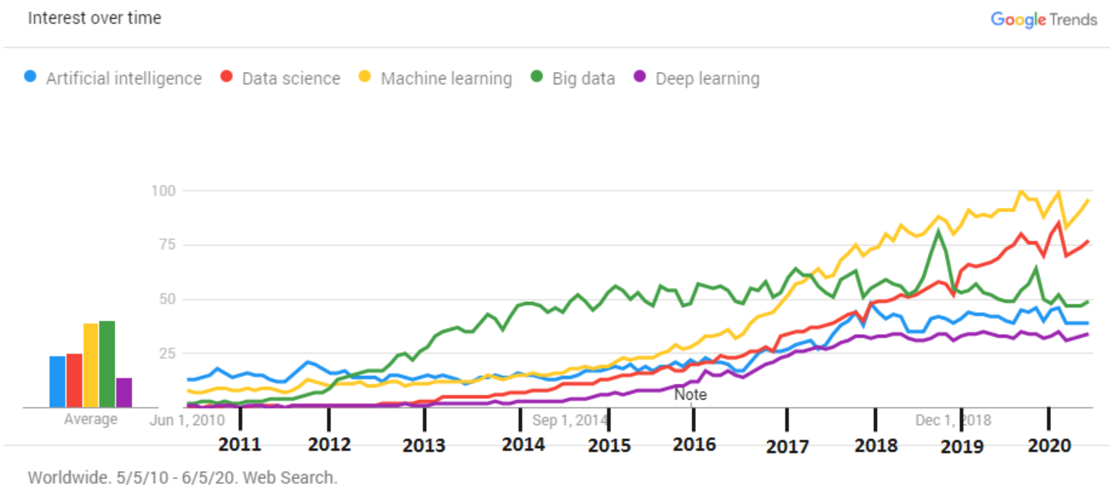


Figure 1.1: Search interest in data science related topics in the period time 2010-2020.[1]

Tech companies are increasingly building large Machine-Learning-as-a-Service (MLaaS) on the Cloud for model training and inference serving. Generally, an ML lifecycle has two distinct phases – training and inference. In a typical MLaaS workflow, developers design and train ML models offline; the training phase is usually characterized by large datasets, long-running hyperparameter searches, dedicated hardware resource usage, and no completion deadlines. The trained models are then published in the Cloud to provide online inference services (inference phase), typically running in containers that can be queried by various end-user applications to make predictions for given inputs [3]. Being user-facing, inference serving requires cost-effective systems that render predictions with latency or accuracy constraints while handling unpredictable and bursty request arrivals.

Inference serving systems face a number of challenges due to the following factors.

[1] https://towardsdatascience.com/has-interest-in-data-science-peaked-already-437648d7f408

a) Diverse application requirements. Applications issue queries that differ in throughput, latency, cost, accuracy, and even privacy requirements. For example, for queries to the same object recognition model, an autonomous vehicle, which continuously receives data from multiple cameras, requires inference in real time, while other applications, such as a production line manufacturing machine, executing a texture inspection or defect detection, may prefer accuracy over latency.
b) Diverse model-variants. Graph optimizers, such as TVM, TensorRT, and methods, such as layer fusion or quantization, produce versions of the same inference model, model-variants, that may differ in inference throughput, latency, memory footprint, and accuracy.
c) Fluctuating resource interference. Depending on the time of day the applications query an inference engine, there might be less or more congestion on the Cloud servers where multiple applications concurrently utilize the shared Cloud resources. On top of that, Cloud providers tend to co-locate applications in shared physical servers to maximize the resource utilization of their infrastructure and reduce their operational cost, which can further increase the congestion on the shared resources.

As a result, together these factors make it difficult for Cloud providers to guarantee an application-specific SLA for each inference query, while also maximizing the resource efficiency of their infrastructures. Simple scheduling approaches that deploy inference engines in the Cloud with a set number of available resources, without considering the unpredictability of the type and amount of user requests throughout the day and the resource load they could create in the Cloud infrastructure, may result either in an over-provision of resources to the application, leading to increased operational costs at the Cloud scale, or an under-provision of resources to the application that leads to huge QoS violations of the applications. What is more, existing inference serving systems require developers to identify the model-variant that can meet diverse performance, accuracy, and cost requirements of applications. However, generating and leveraging these variants requires a substantial understanding of the frameworks, model graph optimizers, and characteristics of hardware architectures, thus limiting the variants an application developer can leverage. As described above, one can use the same object recognition model for several applications, but selecting the appropriate model variant depends on the requirements of an application.

Overview

To address the emerging challenges of ML scheduling, in this thesis, we design and implement an interference and resource aware, predictive scheduling framework, for ML inference engines, that is capable of efficiently utilizing CPU resources to satisfy QoS constraints. Our framework considers the effect of resource interference in the cloud by leveraging low-level system metrics to train and utilize an ML Regression model to predict the Queries per Second (QPS) that an inference engine will achieve, based on the current load and resource utilization. Our scheduling framework aims to serve the inference engine with a parallelism level that results in a performance that meets the required QoS constraint, while being as little resource intensive to the system as possible. We also present a model-less scheduling framework which introduces a model-less interface, where developers need to specify only the ML task that they want to execute (image classification, object detection) and the high-level performance they require as a target QoS. The model-less scheduling framework selects the best Inference Engine, from a task-specific pool of registered, trained Inference models, with the appropriate parallelism level to accomplish the task in the least resource intensive way, while aiming to satisfy the QoS constraint under the current level of interference conditions in the system.

We show that our scheduling framework utilizes a moderate amount of CPU resources, dependent on the target QoS and resource load, to violate QoS constraints, on average, 1.8x less often, compared to the max CPU utilization inference serving system, and 3.1x less often, compared to the min CPU utilization inference serving system, and with a performance variability that is better concentrated around the target QoS, for a variety of interference scenarios and different QoS constraints. What is more, as the QoS constraints change, the model-less scheduling framework retains a similar, on average, overall performance and resource utilization, to the best performing, most efficient ML inference engine each time, resulting, on average, in 1.5x less violations of the QoS constraints and 1.4x less CPU utilization, compared to the model-specific scheduling framework.

The rest of the thesis is organized as follows. Chapter 2 presents related work, Chapter 3 describes the background of Kubernetes, the container orchestrator we used, the MLPerf Inference benchmark suite and the iBench workload suite that were used for the evaluation of our scheduling framework, and a summary of some ML Regression models. Chapter 4 presents our experimental infrastructure, and the characterization of the MLPerf inference engines, Chapter 5 proposes our design to efficiently utilize CPU resources for ML tasks, while satisfying QoS constraints, and Chapter 6 presents experimental results. Chapter 7 concludes the thesis.

Chapter 2

Related Work

2.1 INFaaS: Automated Model-less Inference Serving

Despite existing work in machine learning inference serving, ease-of-use and cost efficiency remain challenges at large scales. Developers must manually search through thousands of model-variants – versions of already-trained models that differ in hardware, resource footprints, latencies, costs, and accuracies – to meet the diverse application requirements. Since requirements, query load, and applications themselves evolve over time, these decisions need to be made dynamically for each inference query to avoid excessive costs through naïve autoscaling.

This paper [1] introduces INFaaS, an automated model-less system for distributed inference serving, where developers simply specify the high-level performance, cost and accuracy requirements for their applications without needing to specify a specific model-variant for each query. INFaaS generates model-variants from already trained registered models, and efficiently navigates the large trade-off space of model-variants on behalf of developers to meet application-specific objectives:

(a) For each query, it selects a model, hardware architecture, and model optimizations. INFaaS tracks the dynamic status of variants (e.g., overloaded or interfered) using a state machine, to efficiently select the right variant for each query to meet the application requirements.

(b) It combines VM-level horizontal autoscaling with model-level autoscaling, where multiple, different model variants are used to serve queries within each machine, so as to be able to dynamically react to the changing application requirements and request patterns.

By leveraging diverse variants and sharing hardware resources across models, INFaaS achieves 1.3x higher throughput, violates latency objectives 1.6x less often, and saves up to 21.6x in cost (8.5x on average) compared to state-of-the-art inference serving systems on AWS EC2.

2.2 Serving Heterogeneous Machine Learning Models on Multi-GPU Servers with Spatio-Temporal Sharing

As machine learning (ML) techniques are applied to a widening range of applications, high throughput ML inference serving has become critical for online services. Such ML inference servers with multiple GPUs must address new challenges in the ML scheduler design. First, they must provide a bounded latency for each inference query to support a consistent service-level objective (SLO). Second, they must be able to serve multiple heterogeneous ML models in a system, as cloud-based consolidation improves system utilization.

To address the two requirements of ML inference servers, this paper [4] proposes a new inference scheduling framework for multi-model ML inference servers. The paper shows that with SLO constraints, GPUs with growing parallelism are not fully utilized for ML inference tasks. To maximize the resource efficiency of GPUs, a key mechanism proposed in this paper is to exploit hardware support for spatial partitioning of GPUs resources.

A) With spatio-temporal sharing, a new abstraction layer of GPU resources is created with configurable GPU resources. The scheduler assigns requests to virtual GPUs, called gpulets, with the most effective amount of resources.
B) The scheduler explores the three-dimensional search space with different batch sizes, temporal sharing, and spatial sharing efficiently. To minimize the cost for cloud-based inference servers, the scheduling framework for gpulets auto-scales the required number of GPUs for a given workload.
C) To consider the potential interference overheads when two ML tasks are running concurrently by spatially sharing a GPU, the scheduling decision is made with an interference prediction model.

The authors' prototype implementation of four GPUs proves that the proposed spatio-temporal scheduling with gpulets enhances throughput by 61.7% on average compared to the prior temporal scheduler that does not partition GPU resources, while satisfying SLOs.

2.3 Deep Learning Workload Scheduling in GPU Datacenters: Taxonomy, Challenges and Vision

Deep learning (DL) shows its prosperity in a wide variety of fields. The development of a DL model is a time-consuming and resource-intensive procedure. Hence, dedicated GPU accelerators have been collectively constructed into a GPU datacenter. An efficient scheduler design for such GPU datacenter is crucially important to reduce the operational cost, achieve high performance for each individual workload, high resource utilization for the entire dataset, high fairness among different users, and improve inference latency and accuracy. However, traditional approaches designed for high-performance computing (HPC) or big data workloads cannot support DL workloads to fully utilize the GPU resources. Recently, substantial schedulers have been proposed to tailor for DL workloads in GPU datacenters, although most of them are designed in an ad-hoc way for some specific objectives.

The authors perform an in-depth analysis about the characteristics of DL workloads and identify the inherent challenges for designing a satisfactory scheduler to manage various DL workloads and resources in GPU datacenters. This paper [5] also surveys existing research efforts for both DL training and inference workloads. It primarily presents how existing schedulers facilitate the respective workloads from the scheduling objectives and resource consumption features. Finally, the authors conclude the limitations and implications from existing designs, which can shed new light on several possible, promising, future research directions of DL scheduler designs in GPU datacenters.

2.4 Morphling: Fast, Near-Optimal Auto-Configuration for Cloud-Native Model Serving

Machine learning models are widely deployed in production cloud to provide online inference services. Efficiently deploying inference services requires careful tuning of hardware and runtime configurations (e.g., GPU type, GPU memory, batch size), which can significantly improve the model serving performance and reduce cost. However, existing autoconfiguration approaches for general workloads, such as Bayesian optimization and white-box prediction, are inefficient in navigating the high-dimensional configuration space of model serving, incurring high sampling cost.

The authors of the paper [3] present Morphling, a fast, near-optimal auto-configuration framework for cloud-native model serving. Morphling employs model-agnostic meta-learning to navigate the large configuration space of an inference service. It trains a metamodel offline to capture the general performance trend under varying inference configurations. Morphling quickly adapts the metamodel to a new inference service by sampling a small number of configurations and uses it to find the optimal one. Morphling is implemented as an easy-to-use, auto-configuration service in Kubernetes, and its performance is evaluated with popular CV and NLP models, as well as with the production inference services in Alibaba. Compared with existing approaches, Morphling reduces the median search cost by 3×-22×, quickly converging to the optimal configuration by sampling only 30 candidates in a large search space consisting of 720 options.

Chapter 3

Background

In this chapter, we provide background information on Kubernetes, a container orchestrator. Furthermore, we present the MLPerf Inference Benchmark suite, as well as the iBench workload suite that we will be using in our experiments. Finally, we introduce the basic concepts of Machine Learning (ML), following them with an analysis of some Machine Learning Regression algorithms.

3.1 Kubernetes, a Container Orchestrator

In this section, we analyze some of the basic concepts of Kubernetes, a container orchestration system.

3.1.1 Docker containers and Orchestration

Virtualization technology increases efficiency in data centers by enabling servers to run multiple operating systems and applications with different requirements and dependencies. Server consolidation has been the focus of virtualization, requiring hardware abstraction to create an environment that can run multiple operating systems. Applications run on virtual machines abstracted away from the hardware. As shown in figure 3.1, Virtual Machines on the left are created on the top of a hypervisor. In Virtual Machines, a complete Operating System is installed. As a result, every VM acts like a guest host. On the other hand, containers, presented on the right, include a container engine, which creates and manages containers. Note that virtualization via containers is also known as containerization. As shown, containerization technology runs multiple containers on a common underlying kernel, which are abstracted away into logical partitions. Linux containers with the docker packaging format allow a user to bundle application code with its runtime dependencies, and deploy in a container. A frequently asked question is if someone should use Virtual Machines or containers for his infrastructure setup.

In the following subsections, those two technologies are described in more detail.

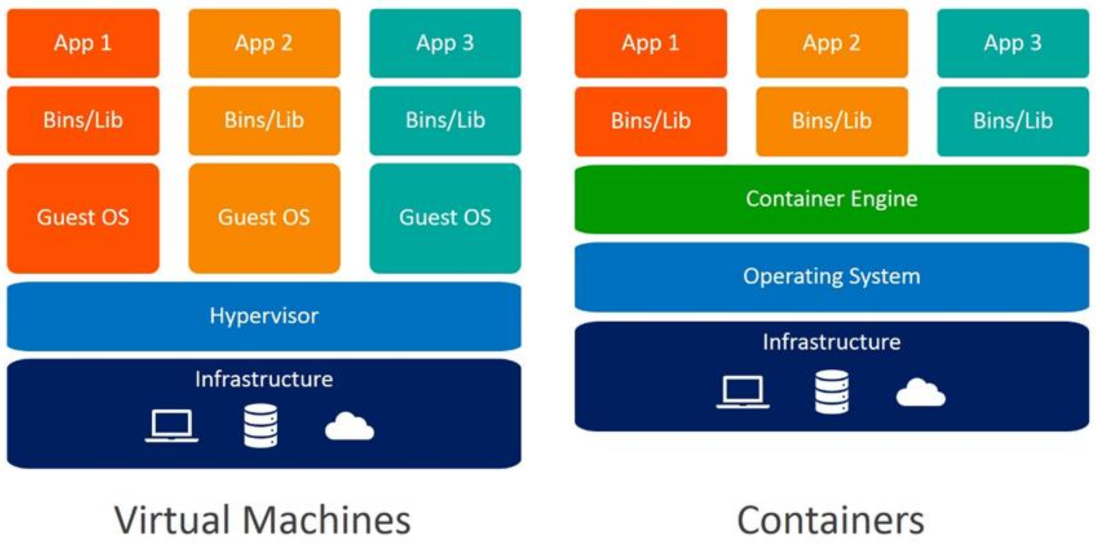


Figure 3.1: Virtual Machines and Containers[2]

Virtual Machines

Virtual Machines (VMs) provide a virtualized hardware environment where a guest OS can run one or more applications. They enable users to create multiple OS instances over the same machine using a hypervisor. Users have the flexibility to allocate CPU, Memory and Disk resources into different VMs. This technology unbounds applications from the machine installed OS. Virtualization has matured to include many resilient capabilities, such as live migration, high availability, SDN, and storage integration which, to date, are not as mature with containerization. Virtualization also provides a higher level of security by running the workload inside a guest operating system that is completely isolated from the host operating system.

Containers

A container is a standard unit of software that packages up code and all its dependencies, so the application runs quickly and reliably from one computing environment to another. A Docker [6] container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings. Container images become containers at runtime, and in the case of Docker containers - images become containers when they run on Docker Engine. Available for both Linux and Windows-based applications, containerized software will always run the same, regardless of the infrastructure. Containers isolate software from its environment and ensure that it works uniformly despite differences, for instance, between development and staging. Containers technology is:

- *Standard:* Docker created the industry standard for containers, so they could be portable anywhere.

[2] https://www.bmc.com/blogs/containers-vs-virtual-machines/

- *Lightweight:* Containers share the machine's OS system kernel and therefore do not require an OS per application, driving higher server efficiency and reducing server and licensing costs. The overhead of booting, managing and maintaining a guest OS environment is avoided. Their lightweight nature leads towards greater start-up speed.

- *Agile application creation and deployment:* Increased ease and efficiency of container image creation and deployment with quick and easy rollbacks (due to image immutability). In fact, it is the application packaging and deployment capability that is revolutionizing DevOps by providing the capability for developers and operations to work side by side, enabling continuous development, integration and deployment. At the same time, environment consistency across development, testing and production is provided, as it runs the same on a laptop as it does in the cloud.

- *Resource isolation:* Containers can be deployed with a fixed amount of resources available. Such techniques control and prevent greedy resources usage.

Orchestration

The answer to the previous question about which virtualization technology is better to use is that they should both be used. They are in fact complementary technologies. Containers support VM-like separation of concerns but with far less overhead and far greater flexibility. As a result, containers have reshaped the way people think about developing, deploying, and maintaining software. In such a hybrid containerized architecture, the different services that constitute an application are packaged into separate containers and deployed across a cluster of virtual machines as illustrated in figure 3.2.

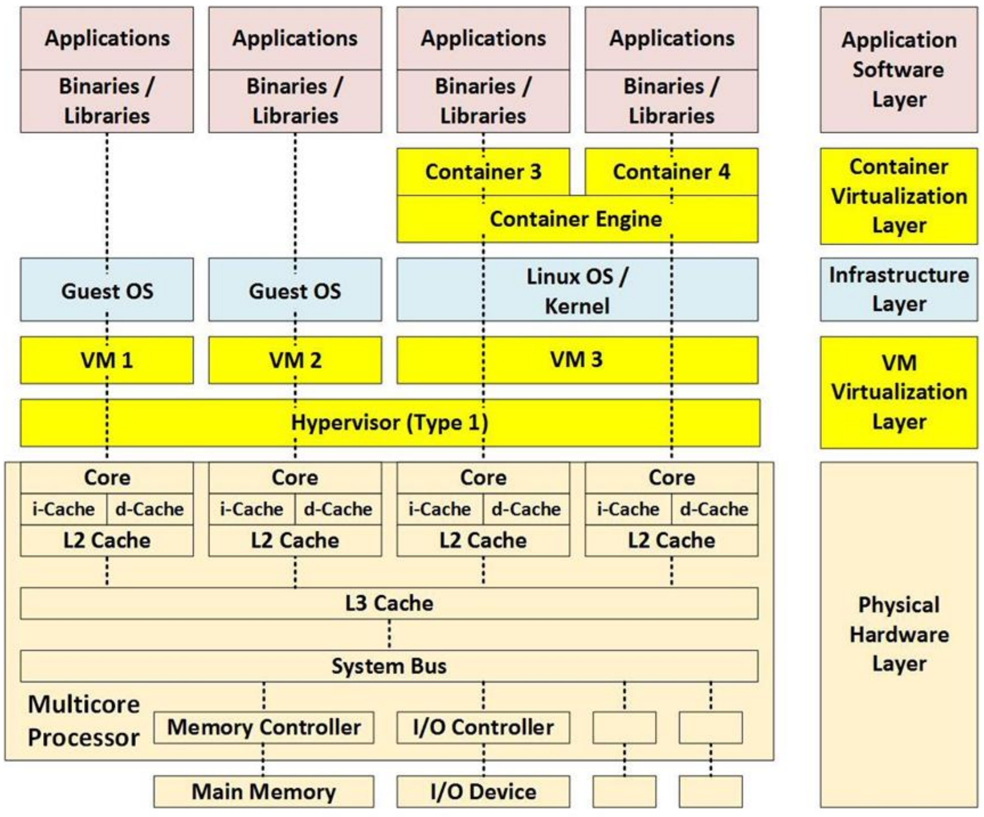


Figure 3.2: Hybrid Containerized Architecture[3]

However, such an architecture highlights the need for container orchestration, a tool that automates the deployment, management, scaling, networking, and availability of container-based applications.

This is where Kubernetes [7] comes in. Large, distributed containerized applications can become increasingly difficult to coordinate. By making containerized applications dramatically easier to manage at scale, Kubernetes has become a key part of the container revolution. It is a portable, extensible platform that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available. Google open-sourced the Kubernetes project in 2014. Kubernetes builds upon a decade and a half of experience that Google has with running production workloads at scale, combined with best-of-breed ideas and practices from the community. In the following sections we describe the different components of that container orchestrator.

3.1.2 Kubernetes Master Node(s) Components

Master components provide the cluster's control plane. Master components make global decisions about the cluster (for example, scheduling), and they detect and respond to cluster events (for example, starting up a new pod when a replication controller's replicas field is unsatisfied). The basic Kubernetes master components are listed below.

- *kube-apiserver:* Component on the master that exposes the Kubernetes API. It is the front-end for the Kubernetes control plane.

- *etcd:* Consistent and highly-available key value store used as Kubernetes' backing store for all cluster data.

- *kube-scheduler:* Component on the master that watches newly created pods that have no node assigned, and selects a node for them to run on.

- *kube-controller-manager:* Component on the master that runs controllers. Logically, each controller is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process.

3.1.3 Kubernetes Worker Node(s) Components

Node Components run on every node as agents maintaining running pods and providing the Kubernetes runtime environment.

- *kubelet:* An agent that runs on each node in the cluster. It makes sure that containers are running in a pod. The kubelet takes a set of PodSpecs that are provided through various mechanisms and ensures that the containers described in those PodSpecs are running and healthy.

[3] https://insights.sei.cmu.edu/sei_blog/2017/09/virtualization-via-containers.html

- *kube-proxy:* A network proxy that runs on each node in the cluster. It enables the Kubernetes service abstraction by maintaining network rules on the host and performing connection forwarding. Kube-proxy is responsible for request forwarding. It allows TCP and UDP stream forwarding or round robin TCP and UDP forwarding across a set of backend functions.

- *Container Runtime:* The container runtime (e.g., Docker) is the software that is responsible for running containers.

Other Important Addons

- *DNS:* Cluster DNS is a DNS server, in addition to the other DNS server(s) in your environment, which serves DNS records for Kubernetes services. Containers started by Kubernetes automatically include this DNS server in their DNS searches.

3.1.4 Kubernetes Architecture

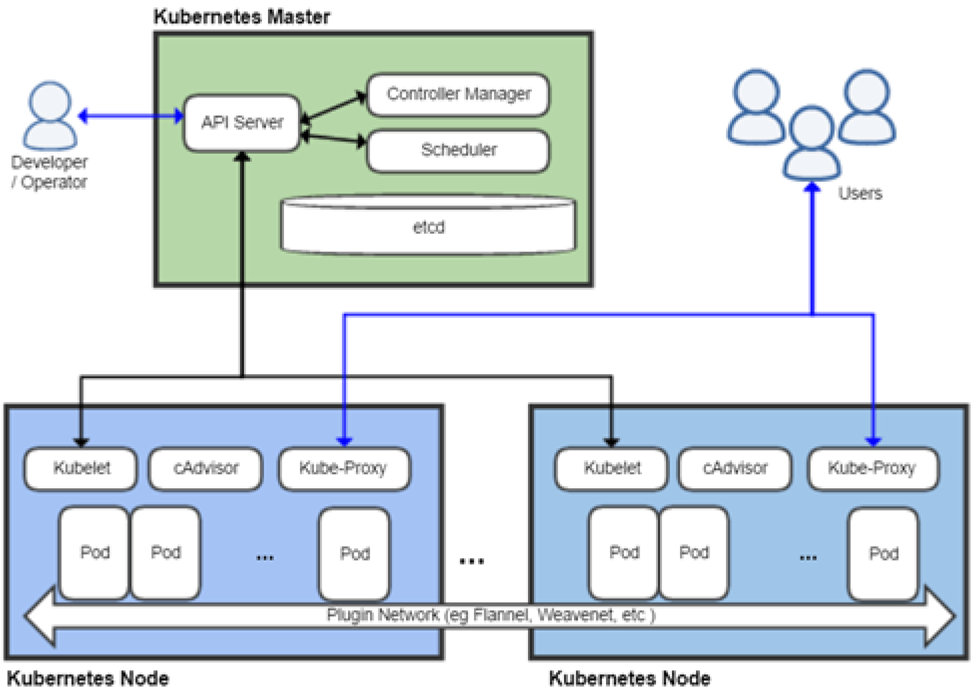


Figure 3.3: Kubernetes Architecture[4]

Kubernetes's architecture makes use of various concepts and abstractions. Some of these are variations on existing, familiar notions, but others are specific to Kubernetes. As illustrated in 3.3 and described before, a Kubernetes cluster is comprised of Nodes. Those nodes are

[4] https://en.wikipedia.org/wiki/Kubernetes

separated into two groups, either Master or Worker nodes. Workloads are executed in Worker Nodes.

Cluster

The highest-level Kubernetes abstraction, the cluster illustrated in figure 3.4, refers to the group of machines running Kubernetes (itself a clustered application) and the containers managed by it. A Kubernetes cluster must have a master, the brain of the system, the node that commands and controls all the other Kubernetes machines in the cluster. A highly available Kubernetes cluster replicates the master's facilities across multiple machines. But only one master at a time runs the job scheduler and controller-manager. The cluster can be set up locally or in the cloud. Most Cloud providers provide a ready-to-use Kubernetes solution.

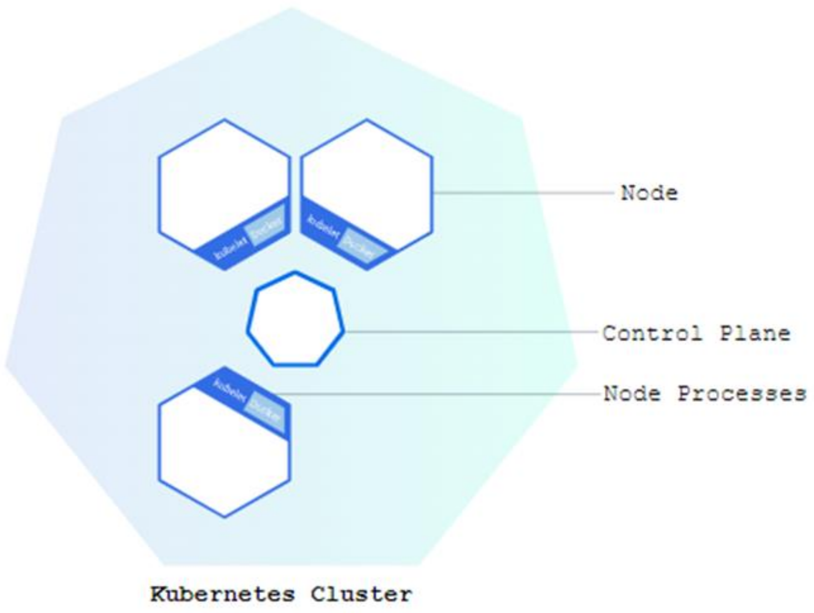


Figure 3.4: Cluster-Node abstraction level[5]

Nodes

Each cluster contains Kubernetes nodes. Nodes might be physical machines or VMs. Again, the idea is abstraction: Whatever the application is running on, Kubernetes handles deployment on that substrate. These Nodes can be either Master Nodes or Worker Nodes. A node with its components is presented in figure 3.5.

[5] https://kubernetes.io/fr/docs/tutorials/kubernetes-basics/create-cluster/cluster-intro/

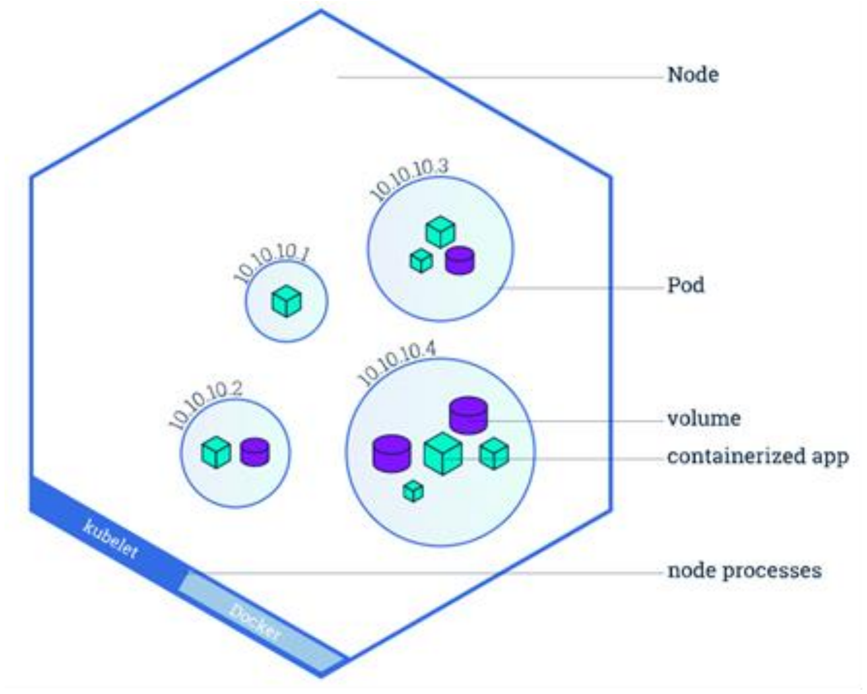


Figure 3.5: Node-Pod-Container abstraction levels[6]

Pods

Nodes run pods, the most basic Kubernetes objects that can be created or managed. Each pod represents a single instance of an application or running process in Kubernetes, and consists of one or more containers as shown in figure 3.5. Kubernetes starts, stops, and replicates all containers in a pod as a group. Pods keep the user's attention on the application, rather than on the application, rather than on the containers themselves. Details about how Kubernetes needs to be configured, from the state of pods on up, are kept in etcd (distributed key-value store).

Pods are created and destroyed on nodes as needed to conform to the desired state specified by the user in the pod definition. Kubernetes provides an abstraction called a controller for dealing with the logistics of how pods are spun up, rolled out, and spun down. Controllers come in a few different flavors depending on the kind of application being managed. For instance, the recently introduced "StatefulSet" controller is used to deal with applications that need persistent state. Another kind of controller, the deployment, is used to scale an app up or down, update an app to a new version, or roll back an app to a known-good version if there is a problem. Also, a deployment will try to reschedule any failed pods. Finally, a deployment tries to provide a guarantee that the required number of pods are running on the cluster.

Deployments

As it is described in Kubernetes documentation, a desired state is described in a Deployment, and the Deployment controller changes the actual state to the desired state at a controlled rate. Deployments are defined to create new ReplicaSets, or to remove existing Deployments and

[6] https://kubernetes.io/docs/tutorials/kubernetes-basics/explore/explore-intro/

adopt all their resources with new Deployments. This object offered the easily manageable scalability, so as to increase or decrease accordingly the required stress levels, just by changing the replicas of the pods created.

Jobs

A Job creates one or more Pods and ensures that a specified number of them successfully terminate. As pods successfully complete, the Job tracks the successful completions. When a specified number of successful completions is reached, the task (i.e., Job) is complete. Deleting a Job will clean up the Pods it created.

A simple case is to create one Job object in order to reliably run one Pod to completion. The Job object will start a new Pod if the first Pod fails or is deleted (for example due to a node hardware failure or a node reboot). A Job can be used to run multiple Pods in parallel.

Volumes

File systems in the Kubernetes container provide ephemeral storage, by default. This means that a restart of the pod will wipe out any data on such containers, and therefore, this form of storage is quite limiting in anything but trivial applications. A Kubernetes Volume provides persistent storage that exists for the lifetime of the pod itself. This storage can also be used as shared disk space for containers within the pod. Volumes are mounted at specific mount points within the container, which are defined by the pod configuration, and cannot mount onto other volumes or link to other volumes. The same volume can be mounted at different points in the file system tree by different containers.

Services

Kubernetes Pods are mortal. They are born and when they die, they are not resurrected. If you use a Deployment to run your app, it can create and destroy Pods dynamically (e.g., when scaling out or in). Each Pod gets its own IP address, however the set of Pods for a Deployment running in one moment in time could be different from the set of Pods running that application a moment later. This leads to the following problem: if a set of Pods (call them “backends”) provides functionality to other Pods (call them “frontends”) inside your cluster, how do those frontends find out and keep track of which IP address to connect to, so that the frontend can use the backend part of the workload? A Service is an abstract way to expose an application running on a set of Pods as a network service. Kubernetes gives pods their own IP addresses and a single DNS name for a set of pods and can load-balance across them.

3.1.5 Kubernetes Resources

When the user specifies a Pod, he can optionally specify how much CPU and memory (RAM) each container needs. When containers have resource requests specified, the scheduler can make better decisions about which nodes to place Pods on. And when Containers have their limits specified, contention for resources on a node can be handled in a specified manner[7].

Resource Types: *CPU* and *memory* are each a resource type. A resource type has a base unit. CPU is specified in units of cores, and memory is specified in units of bytes. CPU and memory are collectively referred to as compute resources, or just resources. Compute resources are measurable quantities that can be requested, allocated, and consumed. They are distinct from API resources. API resources, such as Pods and Services are objects that can be read and modified through the Kubernetes API server.

Each Container of a Pod can specify one or more of the following:

- spec.containers[].resources.limits.cpu
- spec.containers[].resources.limits.memory
- spec.containers[].resources.requests.cpu
- spec.containers[].resources.requests.memory

Although requests and limits can only be specified on individual Containers, it is convenient to talk about Pod resource requests and limits. A Pod resource request/limit for a particular resource type is the sum of the resource requests/limits of that type for each Container in the Pod.

Meaning of CPU and Memory: *Limits* and *requests* in CPU resources are measured in cpu units. One CPU in Kubernetes is equivalent to 1 vCPU or 1 Hyperthread on a bare-metal Intel processor. Also, fractional requests are allowed. For example, a request of 0.5 cpu (or 500m which can be read as five hundred millicpu), allocates half of a CPU. CPU is always requested as an absolute quantity, never as a relative quantity; 0.5 is the same amount of CPU on a single-core, dual-core, or a 48-core machine. Regarding the Memory's requests and limits, they are measured in bytes. Someone can express memory as a plain integer, or as a fixed-point integer. Also, the user can use the power-of-two equivalents: Ei, Pi, Ti, Gi, Mi, Ki.

These requests and limits are passed to the container runtime, when the kubelet starts a container of a Pod. When using Docker, the *–cpu-shares* and *–memory* flags are used accordingly.

When you create a Pod, the Kubernetes scheduler selects a node for the Pod to run on. Each node has a maximum capacity for each of the resource types: the amount of CPU and memory it can provide for Pods. The scheduler ensures that, for each resource type, the sum of the resource requests of the scheduled containers is less than the capacity of the node. Note that although actual memory or CPU resource usage on nodes is very low, the scheduler still refuses to place a Pod on a node if the capacity check fails. This protects against a resource shortage on a node when resource usage later increases, for example, during a daily peak in request rate.

[7] https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/

3.1.6 Kubernetes Scheduling

The Kubernetes Scheduler is a core component of Kubernetes: After a user or a controller creates a Pod, the Kubernetes Scheduler, monitoring the Object Store for unassigned Pods, will assign the Pod to a Node. Then, the kubelet, monitoring the Object Store for assigned Pods, will execute the Pod.

For each unscheduled Pod, the Kubernetes scheduler tries to find a node across the cluster according to a set of rules. There are two steps before a destination node of a Pod is chosen. The first step is filtering all the nodes (*Node Filtering*) and the second is ranking the remaining nodes (*Node Prioritizing*) to find a best fit for the Pod. In the *Node Filtering* phase, the scheduler determines the set of feasible placements, which is the set of nodes that meet a set of given constraints. All filter functions, also called predicates, must yield true for the Node to host the Pod. In the *Node Prioritizing* phase, with only the feasible Nodes remaining, Kubernetes scheduler using a set of predefined rating functions, determines the viability of each Node. The Pod will be scheduled in the one with the highest viability.

Kubernetes scheduler uses this technique for two reasons. Firstly, it needs to make sure that no pod will be scheduled in a Node that is unable to handle it, taking into account its Quality of Service (QoS), which can be declared with a few configurations in the yaml file that creates the pod. Secondly, that way it will run the second part of the algorithm (prioritization functions) across a much less set of nodes, which will consume less system resources and less time.

In the following image we can see the steps of the scheduling algorithm.

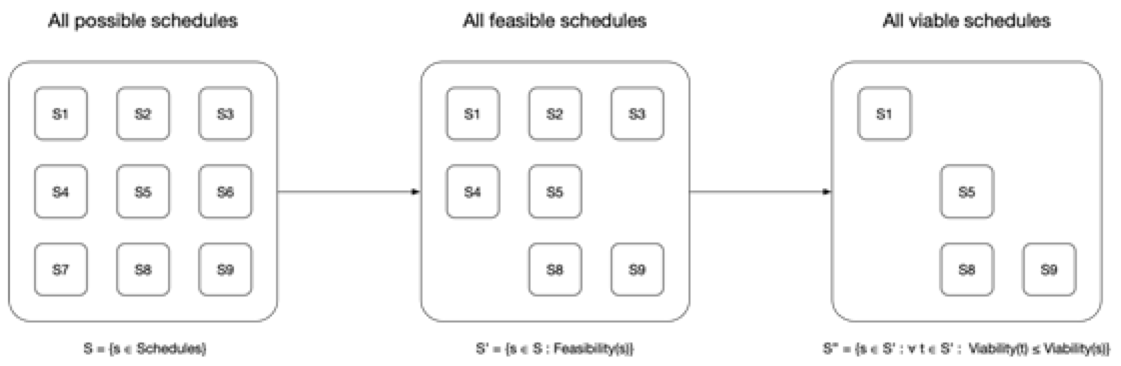


Figure 3.6: Node filtering and ranking[8]

3.2 MLPerf™ Inference Benchmark Suite

The MLPerf benchmark suites are the industry-standard for measuring machine learning system performance. Each benchmark suite is focused on different types of systems and workloads. Driven by more than 30 organizations as well as more than 200 ML engineers and practitioners, MLPerf prescribes a set of rules and best practices to ensure comparability across systems with wildly differing architectures.

[8] https://medium.com/@dominik.tornow/the-kubernetes-scheduler-cd429abac02f

Machine-learning (ML) hardware and software system demand is burgeoning. Driven by ML applications, the number of different ML inference systems has exploded. Over 100 organizations are building ML inference chips, and the systems that incorporate existing models span at least three orders of magnitude in power consumption and five orders of magnitude in performance; they range from embedded devices to data-center solutions. Fueling the hardware are a dozen or more software frameworks and libraries. The myriad combinations of ML hardware and ML software make assessing ML-system performance in an architecture-neutral, representative, and reproducible manner challenging. There is a clear need for industry-wide standard ML benchmarking and evaluation criteria. MLPerf Inference answers that call [8].

MLPerf Inference is a benchmark suite that evaluates ML inference systems (datacenter and edge), by measuring how fast they process inputs and produce results using/running a trained model in a variety of deployment scenarios. Examples tasks include recommendation, question answering, speech-to-text, object detection, and image recognition.

MLPerf is a part of the MLCommons™ Association. MLCommons is an open engineering consortium that promotes the acceleration of machine learning innovation. It provides benchmarks and metrics, datasets and models, and best practices.[9]

3.2.1 MLPerf Inference Benchmarks for Image Classification and Object Detection Tasks

Computer vision is a field of artificial intelligence (AI) that enables computers and systems to derive meaningful information from digital images, videos and other visual inputs — and take actions or make recommendations based on that information.[10]

In this field, two important tasks to evaluate the performance of a ML inference system, using the MLPerf Inference benchmark suite, are image classification and object detection [9].

This is the reference implementation for the MLPerf Inference benchmarks. Each MLPerf Inference benchmark is defined by a pre-trained model, a backend framework and a dataset.

Area	Task	**model**	**framework**	**accuracy**	**dataset**	**precision**
Vision	Image Classification	resnet50-v1.5	tensorflow	76.456%	imagenet2012 validation	fp32
Vision	Image Classification	resnet50-v1.5	onnx	76.456%	imagenet2012 validation	fp32
Vision	Image Classification	mobilenet-v1	tensorflow	71.676%	imagenet2012 validation	fp32

[9] https://infohub.delltechnologies.com/p/introduction-to-mlperf-tm-inference-v1-0-performance-with-dell-emc-servers/
[10] https://www.ibm.com/topics/computer-vision

Vision	Image Classification	mobilenet-v1 quantized	tensorflow	70.694%	imagenet2012 validation	int8
Vision	Image Classification	mobilenet-v1	tflite	71.676%	imagenet2012 validation	fp32
Vision	Image Classification	mobilenet-v1	onnx	71.676%	imagenet2012 validation	fp32
Vision	Object Detection	ssd-mobilenet 300x300	tensorflow	mAP 0.23	coco resized to 300x300	fp32
Vision	Object Detection	ssd-mobilenet 300x300 quantized finetuned	tensorflow	mAP 0.23594	coco resized to 300x300	int8
Vision	Object Detection	ssd-mobilenet 300x300 symmetrically quantized finetuned	tensorflow	mAP 0.234	coco resized to 300x300	int8
Vision	Object Detection	ssd-mobilenet 300x300	onnx	mAP 0.23	coco resized to 300x300	fp32

Table 3.1: Supported Models of the MLPerf Inference Benchmark suite for Image Classification and Object Detection Tasks[11]

The key component of the MLPerf Inference Benchmarks is the Load Generator [10]. The LoadGen is a reusable module that efficiently and fairly measures the performance of inference systems. It generates traffic for scenarios as formulated by a diverse set of experts in the MLCommons working group. The scenarios emulate the workloads seen in mobile devices, autonomous vehicles, robotics, and cloud-based setups. Although the LoadGen is not model or dataset aware, its strength is in its reusability with logic that is.[12]

In the MLPerf inference evaluation framework, the LoadGen load generator sends inference queries to the system under test (SUT). The SUT uses a backend (for example, ONNX Runtime, TensorFlow, TFLite or PyTorch) to perform inferencing and returns the results to LoadGen.

The following is a diagram of how the LoadGen can be integrated into an inference system, resembling how the used MLPerf reference models are implemented.

[11] https://github.com/mlcommons/inference/tree/master/vision/classification_and_detection

[12] https://github.com/mlcommons/inference/tree/master/loadgen

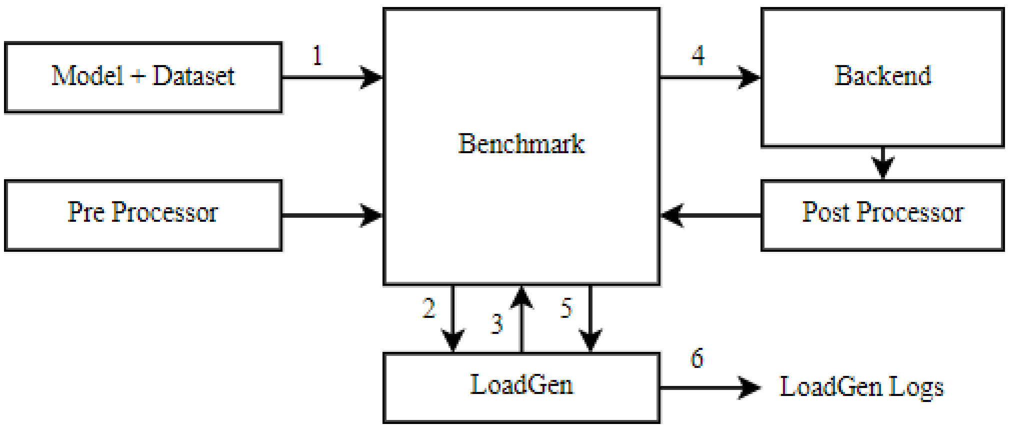


Figure 3.7: LoadGen Integration in MLPerf Inference Benchmarks[13]

As shown in the diagram, the Benchmark knows the model, dataset, and preprocessing (1). The Benchmark hands dataset sample IDs to LoadGen (2). LoadGen starts generating inference queries of sample IDs (3). These queries are translated by the Benchmark to backend requests to perform inferencing (4). The result is post processed and forwarded to LoadGen (5). Finally, LoadGen outputs logs for analysis (6).

In order to enable representative testing of a wide variety of inference platforms and use cases, MLPerf has defined four different inference scenarios as described in the table below. The main differences between these scenarios are based on the particular pattern in which the standard load generator generates and sends inference requests to the SUT, as well as the performance metric that is used:[14]

Scenario	Query Generation	Duration	Samples/query	Latency Constraint	Tail Latency	Performance Metric
Single stream	LoadGen sends next query as soon as SUT completes the previous query	600 seconds	1	None	90%	90%-ile early-stopping latency estimate
Server	LoadGen sends new queries to the SUT according to a Poisson distribution	600 seconds	1	Benchmark specific	99%	Maximum Poisson throughput parameter supported

[13] https://github.com/mlcommons/inference/tree/master/loadgen
[14] https://github.com/mlcommons/inference_policies/blob/master/inference_rules.adoc#scenarios

Offline	LoadGen sends all samples to the SUT at start in a single query	1 query and 600 seconds	At least 24,576	None	N/A	Measured throughput
Multistream	Loadgen sends next query, as soon as SUT completes the previous query	600 seconds	8	None	99%	99%-ile early-stopping latency estimate

Table 3.2: Scenario table

An early stopping criterion allows for runs with a relatively small number of processed queries to be valid, with the penalty that the effective computed percentile will be slightly higher. This penalty counteracts the increased variance inherent to runs with few queries, where there is a higher probability that a particular run will, by chance, report a lower latency than the system should reliably support.

The MLPerf Inference benchmark suite also offers some additional options during testing. Among others, the --backend option defines which backend (currently supported are: tensorflow, onnxruntime, pytorch and tflite) to use, the --scenario {SingleStream,MultiStream,Server,Offline} option defines which scenario to run, the --time option (in seconds) limits the time the benchmark runs, and the --threads option sets the number of worker threads to use during the run (default: the number of processors in the system).

3.3 iBench: Quantifying interference for datacenter applications

Interference between co-scheduled applications is one of the major reasons that causes modern datacenters (DCs) to operate at low utilization. DC operators traditionally side-step interference either by disallowing colocation altogether and providing isolated server instances, or by requiring the users to express resource reservations, which are often exaggerated to counter-balance the unpredictability in the quality of allocated resources. Understanding, reducing and managing interference can significantly impact the manner in which these large-scale systems operate.

This is where iBench comes in, a workload suite that helps quantify the pressure different applications put in various shared resources, and similarly the pressure they can tolerate in these resources, resulting in significant performance and/or efficiency improvements in applications. iBench consists of a set of carefully-crafted benchmarks that induce interference of tunable increasing intensity in shared resources in a multi-core chip that span the CPU cores, cache hierarchy, memory bandwidth and capacity, storage and networking subsystems [11].

The goal of iBench is to identify the shared resources an application creates contention to, and similarly the type and amount of contention the application is sensitive to. For this purpose, all iBench workloads (sources of interference (SoIs)) have tunable intensity that progressively puts more pressure on a specific shared resource until the behavior of the application changes (i.e., performance degrades). SoIs are designed such that their impact to the corresponding resource increases almost linearly with the intensity of the benchmark. Finally, the impact of the different iBench workloads is not overlapping, e.g., the memory bandwidth SoI does not cause significant contention in memory capacity and vice versa.

3.4 Machine Learning (ML)

Machine learning (ML) is a field of inquiry devoted to understanding and building methods that “learn”, that is, methods that leverage data to improve performance on some set of tasks [12]. It is seen, essentially, as a subfield of artificial intelligence (AI). The term “machine learning” was coined in 1959 by Arthur Samuel, an IBM employee and pioneer in the field of computer gaming and artificial intelligence [13]. Also, the synonym “self-teaching computers” was used in this time period [14].

Machine learning algorithms build a model based on sample data, known as training data, in order to make accurate outcome predictions or decisions without being explicitly programmed to do so [15]. Machine learning algorithms are used in a wide variety of applications, such as in medicine, finance and trading, email filtering, speech recognition, and computer vision, where it is difficult or unfeasible to develop conventional algorithms to perform the needed tasks [16].

Machine learning approaches are traditionally divided into three broad categories, which correspond to machine learning paradigms, depending on the nature of the "signal" or "feedback" available to the learning system:

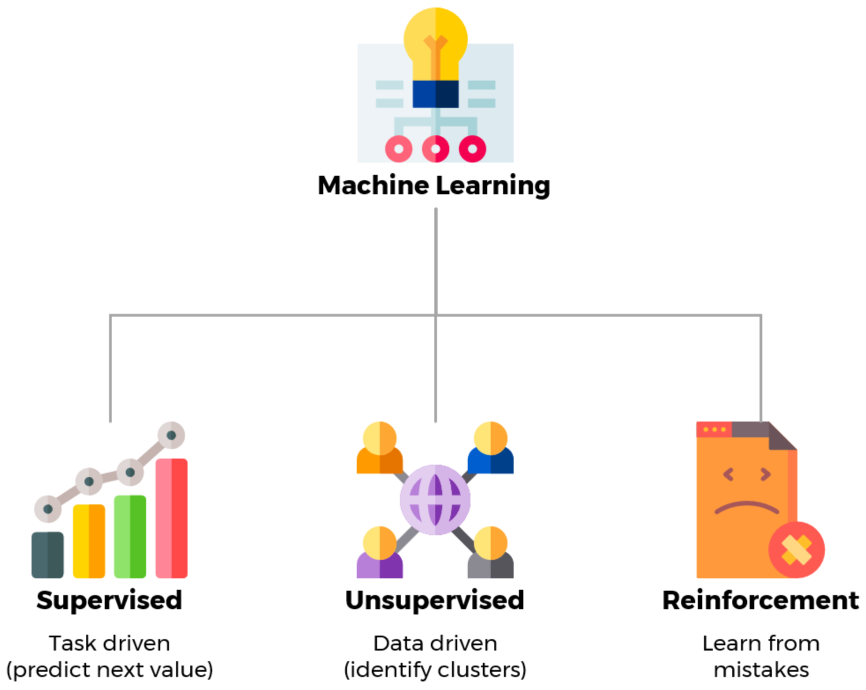


Figure 3.8: The 3 Types of Machine Learning[15]

Supervised learning: Supervised learning algorithms build a mathematical model of a set of training data that contains example inputs and their desired outputs (also called the supervisory signal). In these labelled examples, each data point contains features (covariates) and an associated label. The goal of supervised learning algorithms is learning a general rule, a function, that maps feature vectors (inputs) to labels (outputs), based on example input-output pairs [17]. This inferred function can then be used for mapping new examples. An optimal scenario will allow for the algorithm to correctly determine the class labels for unseen instances that were not a part of the training data.

Types of supervised-learning algorithms include active learning, classification and regression. Classification algorithms are used when the outputs are restricted to a limited set of values, and regression algorithms are used when the outputs may have any numerical value within a range. As an example, for a classification algorithm that filters emails, the input would be an incoming email, and the output would be the name of the folder in which to file the email.

Unsupervised learning: Unsupervised learning algorithms take a set of data that contains only inputs, meaning no labels are given to the learning algorithm, leaving it on its own to find structure in its input, like grouping or clustering of data points. The algorithms, therefore, act on the data without prior training. Instead of responding to feedback, unsupervised learning algorithms identify commonalities in the data and react based on the presence or absence of such commonalities in each new piece of data. Unsupervised learning can be a goal in itself (discovering hidden patterns in data) or a means towards an end (feature learning).

Reinforcement learning: Reinforcement learning is an area of machine learning concerned with how software agents ought to take actions in an environment so as to maximize some notion of cumulative reward. A computer program interacts with a dynamic environment in which it must perform a certain goal (such as driving a vehicle or playing a game against an opponent). As it navigates its problem space, the program is provided feedback that is analogous to rewards, which it tries to maximize [18].

3.4.1 Regression Models in Machine Learning

In statistical modeling, regression analysis is a set of statistical processes for estimating the relationships between a dependent variable (often called the 'outcome' or 'response' variable, or a 'label' in machine learning parlance) and one or more independent variables (often called 'predictors', 'covariates', 'explanatory variables' or 'features'). In the field of machine learning, regression is a supervised learning technique which helps in finding the correlation between variables and enables us to predict the continuous, numeric output variable(s) based on the one or more independent variables. It is mainly used for prediction, forecasting, time series modeling, and determining the causal-effect relationship between variables.

[15] https://hethelinnovation.com/in-a-nutshell/machine-learning-in-a-nutshell/

There are various types of regression analysis which are used in data science and machine learning. Each type has its own importance on different scenarios, but at the core, all the regression methods analyze the effect of the independent variables on dependent variables.[16]

Linear Regression

Linear regression is one of the most basic types of regression in machine learning. The linear regression model consists of a predictor input variable (X-axis) and a dependent output variable (Y-axis) related linearly to each other. If there is only one input variable (x), then such linear regression is called simple linear regression. In case the data involves more than one independent variable, then such linear regression is called multiple linear regression.

The relationship between variables in the linear regression model can be explained using the image below.

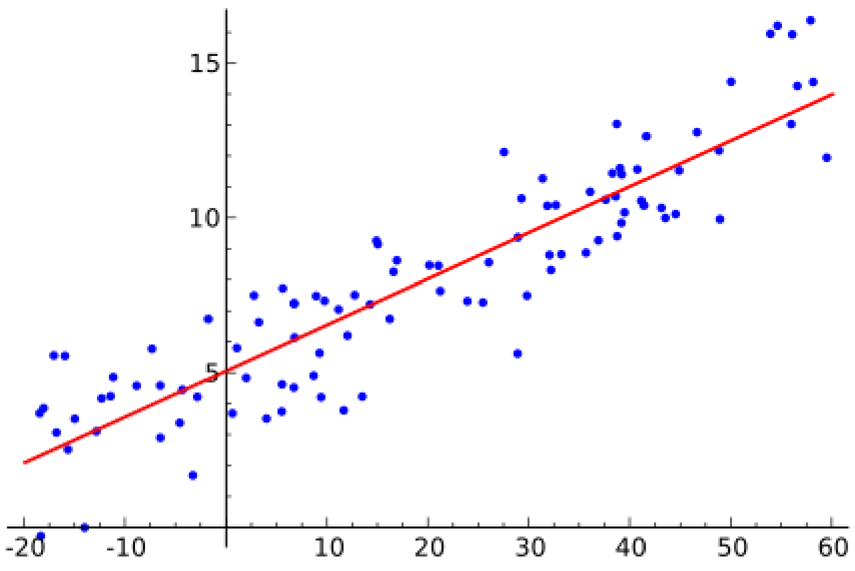


Figure 3.9: Illustration of linear regression on a data set.[17]

Regression models involve the following components:

- The **unknown parameters**, often denoted as a scalar or vector β, with β_0 being the intercept term.
- The **independent variables**, which are observed in data and are often denoted as a vector X_i (where i denotes a row of data).
- The **dependent variable**, which are observed in data and often denoted using the scalar Y_i.
- The **error terms**, which are not directly observed in data and are often denoted using the scalar ε_i.

[16] https://www.javatpoint.com/regression-analysis-in-machine-learning
[17] https://en.wikipedia.org/wiki/Regression_analysis#Linear_regression

Given a data set $\{y_i, x_{i1}, \dots, x_{ip}\}_{i=1}^{n}$ of n statistical units, a linear regression model assumes that the relationship between the dependent variable $\mathbf{y}$ and the p-vector of regressors $\mathbf{x}$ is linear. This relationship is modeled through a disturbance term or error variable ε — an unobserved random variable that adds "noise" to the linear relationship between the dependent variable and regressors. Thus, the model takes the form

$$y_i = \beta_0 + \beta_1 x_{i1} + \cdots + \beta_p x_{ip} + \varepsilon_i = \boldsymbol{x}_i^T \boldsymbol{\beta} + \varepsilon_i \qquad i = 1, \dots, n$$

where ^{T} denotes the transpose, so that $\boldsymbol{x}_i^T \boldsymbol{\beta}$ is the inner product between vectors $\boldsymbol{x_i}$ and $\boldsymbol{\beta}$.

Often these n equations are stacked together and written in matrix notation as

$$\boldsymbol{y} = \boldsymbol{X\beta} + \boldsymbol{\varepsilon}$$

Fitting a linear model to a given data set usually requires estimating the regression coefficients $\boldsymbol{\beta}$ such that the error term $\boldsymbol{\varepsilon} = \boldsymbol{y} - \boldsymbol{X\beta}$ is minimized. It is common to use the sum of squared errors as a measure of **ε** for minimization.

Ridge regression

Ridge regression is a method of estimating the coefficients of multiple-regression models in scenarios where the independent variables are highly correlated [19]. Ridge regression was developed as a possible solution to the imprecision of least square estimators when linear regression models have some multicollinear (highly correlated) independent variables—by creating a ridge regression estimator (RR). This provides a more precise ridge parameters estimate, as its variance and mean square estimator are often smaller than the least square estimators previously derived.

Linear regression is the standard algorithm for regression that assumes a linear relationship between input variables and the target variable. An extension to linear regression invokes adding penalties to the loss function during training that encourages simpler models that have smaller coefficient values. These extensions are referred to as regularized linear regression or penalized linear regression.

In particular, Ridge Regression is a popular type of regularized linear regression that includes an L2 penalty, a small amount of bias added to the model. This has the effect of shrinking the coefficients for those input variables that do not contribute much to the prediction task.[18]

In Linear Regression with a single input variable, the relationship that connects the input variables to the target variable is a line, and with higher dimensions, this relationship can be thought of as a hyperplane. The coefficients of the model are found via an optimization process that seeks to minimize the sum squared error between the predictions ($\hat{y}$) and the expected target values (y).

$$loss = \sum_{i=0}^{n} (y_i - \hat{y}_i)^2$$

[18] https://machinelearningmastery.com/ridge-regression-with-python/

A problem with linear regression is that estimated coefficients of the model can become large, making the model sensitive to inputs and possibly unstable. This is particularly true for problems with few observations (samples) or less samples (n) than input predictors (p) or variables (so-called p >> n problems).

One approach to address the stability of regression models is to change the loss function to include additional costs for a model that has large coefficients. One popular penalty is to penalize a model based on the sum of the squared coefficient values (β). This is called an L2 penalty.

$$L2\_penalty = \sum_{j=0}^{p} \beta_j^2$$

An L2 penalty minimizes the size of all coefficients, although it prevents any coefficients from being removed from the model by not allowing their value to become zero. This penalty can be added to the cost function for linear regression and is referred to as Tikhonov regularization (after the author), L2 regularization or Ridge Regression more generally.

A hyperparameter called "lambda" is used, that controls the weighting of the penalty to the loss function. A default value of 1.0 will fully weight the penalty; a value of 0 excludes the penalty. Very small values of lambda, such as 1e-3 or smaller are common.

$$Ridge\_loss = loss + \lambda * L2\_penalty$$

Lasso Regression

Lasso (least absolute shrinkage and selection operator) Regression is a popular type of regularized linear regression that includes an L1 regularization penalty to the loss function of linear regression during training. This has the effect of shrinking the coefficients for those input variables that do not contribute much to the prediction task. This penalty allows some coefficient values to go to the value of zero, allowing input variables/features to be effectively removed from the model, providing a type of automatic feature selection.[19]

Similarly to Ridge Regression, Lasso improves on the stability of linear regression by changing the loss function to include additional costs for a model that has large coefficients. However, this time, the L1 penalty is used. L1 is a popular penalty that penalizes a model based on the sum of the absolute coefficient values (β). An L1 penalty minimizes the size of all coefficients and even allows some coefficients to be minimized to the value zero, which removes the predictor from the model.

$$L1\_penalty = \sum_{j=0}^{p} |\beta_j|$$

[19] https://machinelearningmastery.com/lasso-regression-with-python/

A consequence of penalizing the absolute values is that some parameters are actually set to 0 for some value of lambda. Thus, the lasso yields models that simultaneously use regularization to improve the model and to conduct feature selection.

$$Lasso\_loss = loss + \lambda * L1\_penalty$$

Lasso Regression is also known as L1 regularization.

Elastic Net Regression

Elastic net is a popular type of regularized linear regression that linearly combines two popular penalties, specifically the L1 and L2 penalty functions of the lasso and ridge methods accordingly.[20]

An L2 penalty minimizes the size of all coefficients, although it prevents any coefficients from being removed from the model, whereas n L1 penalty minimizes the size of all coefficients and allows some coefficients to be minimized to the value zero, effectively removing input features from the model.

Elastic net is a penalized linear regression model that includes both the L1 and L2 penalties during training. A hyperparameter "alpha" is provided to assign how much weight is given to each of the L1 and L2 penalties. Alpha is a value between 0 and 1 and is used to weight the contribution of the L1 penalty and one minus the alpha value is used to weight the L2 penalty.

$$Elastic\_Net\_penalty = \alpha * L1\_penalty + (1 - \alpha) * L2\_penalty$$

The parameter alpha determines the mix of the penalties, and is often pre-chosen on qualitative grounds. For example, an alpha of 0.5 would provide a 50 percent contribution of each penalty to the loss function. An alpha value of 0 gives all weight to the L2 penalty and a value of 1 gives all weight to the L1 penalty.

The benefit is that elastic net allows a balance of both penalties, which can result in better performance than a model with either one or the other penalty on some problems.

$$Elastic\_Net\_loss = loss + \lambda * Elastic\_Net\_penalty$$

Bayesian Linear Regression

Bayesian linear regression is a type of conditional modeling in which the mean of one variable is described by a linear combination of other variables, with the goal of obtaining the posterior probability of the regression coefficients (as well as other parameters describing the distribution of the regressand) and ultimately allowing the out-of-sample prediction of the regressand (often labelled y) conditional on observed values of the regressors (usually X). The simplest and most widely used version of this model is the normal linear model, in which y given X is distributed Gaussian. In this model, and under a particular choice of prior probabilities for the parameters—so-called conjugate priors—the posterior can be found

[20] https://machinelearningmastery.com/elastic-net-regression-in-python/

analytically. With more arbitrarily chosen priors, the posteriors generally have to be approximated.[21]

Bayesian Regression can be very useful when we have insufficient data in the dataset or the data is poorly distributed. The output of a Bayesian Regression model is obtained from a probability distribution, as compared to regular regression techniques where the output is just obtained from a single value of each attribute. The output, 'y' is generated from a normal distribution (where mean and variance are normalized). The aim of Bayesian Linear Regression is not to find the model parameters, but rather to find the "posterior" distribution for the model parameters. Not just the output y, but the model parameters are also assumed to come from a distribution. The expression for Posterior is[22] :

$$Posterior = (Likelihood * Prior)/Normalization$$

where

Posterior: the probability of an event to occur; say, H, given that another event; say, E has already occurred. i.e., P(H | E).

Prior: the probability an event H has occurred prior to another event. i.e., P(H)

Likelihood: a likelihood function in which some parameter variable is marginalized.

Looking at the formula above, we can see that, in contrast to Ordinary Least Square (OLS), we have a posterior distribution for the model parameters which is proportional to the likelihood of the data multiplied by the prior probability of the parameters. As the number of data points increase, the value of likelihood will increase and will become much larger than the prior value. In the case of an infinite number of data points, the values for the parameters converge to the values obtained from OLS. So, we begin our regression process with an initial estimate (the prior value). As we start to cover more data points, our model becomes less wrong. As a result, a large amount of training data is needed to make the model accurate.

Bayesian Regression is particularly well-suited for on-line, stream-based learning (data is received in real-time), as compared to batch-based learning, where we have the entire dataset on our hands before we start training the model. This is because Bayesian Regression doesn't need to store data. However, despite Bayesian linear regression (BLR) being a powerful tool to analyze regression problems, it has some shortcomings. Primarily, the outcome of BLR is highly dependent on our assumptions about data. This means if we make wrong assumptions about data distribution, the generated regression model will go wrong. Another shortcoming of this model is that it assumes constant noise over global data range, which may not reflect the actual noise levels for large datasets, where each region has a different level of noise. A third disadvantage of this method appears when data is not normally distributed. In this case, a sophisticated Bayesian analysis needs to be done to discover the data and the prior distributions with their proper initial settings.[23]

The Bayesian approach can be used with any Regression technique like Linear Regression, Ridge Regression, Lasso Regression, etc.

[21] https://en.wikipedia.org/wiki/Bayesian_linear_regression
[22] https://www.geeksforgeeks.org/implementation-of-bayesian-regression/
[23] https://www.data-automaton.com/2021/03/03/bayesian-linear-regression/

Stochastic Gradient Descent (SGD)

Stochastic gradient descent (often abbreviated SGD) is an iterative method for optimizing an objective function with suitable smoothness properties (e.g., differentiable or subdifferentiable). It can be regarded as a stochastic approximation of gradient descent optimization, since it replaces the actual gradient (calculated from the entire data set) by an estimate thereof (calculated from a randomly selected subset of the data). Especially in high-dimensional optimization problems this reduces the very high computational burden, achieving faster iterations in trade for a lower convergence rate [20].

Stochastic Gradient Descent (SGD) is a simple yet very efficient approach to fitting linear classifiers and regressors under convex loss functions such as (linear) Support Vector Machines and Logistic Regression. Even though SGD has been around in the machine learning community for a long time, it has received a considerable amount of attention just recently in the context of large-scale learning.[24]

SGD has been successfully applied to large-scale and sparse machine learning problems often encountered in text classification and natural language processing. Given that the data is sparse, the classifiers in this module easily scale to problems with more than 10^5 training examples and more than 10^5 features.

Strictly speaking, SGD is merely an optimization technique and does not correspond to a specific family of machine learning models. It is only a way to train a model.

The advantages of Stochastic Gradient Descent are efficiency and ease of implementation (lots of opportunities for code tuning). The disadvantages of Stochastic Gradient Descent include that SGD requires a number of hyperparameters such as the regularization parameter and the number of iterations, and also that SGD is sensitive to feature scaling.

Support Vector Regression (SVR)

Support-vector machines (SVMs, also support-vector networks [21]) are supervised learning models with associated learning algorithms that analyze data for classification analysis, regression analysis and outliers detection. The version of SVM for regression is called support vector regression (SVR) [22]. The model produced by SVR depends only on a subset of the training data, because the cost function for building the model ignores any training data close to the model prediction.

In general SVM, the hyperplane is a separation line between two classes, but in SVR, it is a line which helps to predict the continuous variables and cover most of the datapoints. Boundary lines are the two lines apart from the hyperplane, which create a margin for the datapoints. Lastly, the support vectors are the datapoints which are nearest to the hyperplane and opposite class in SVM.

In SVR, we always try to determine a hyperplane with a maximum margin, so that maximum number of datapoints are covered in that margin. The main goal of SVR is to consider the

[24] https://scikit-learn.org/stable/modules/sgd.html#

maximum datapoints within the boundary lines and the hyperplane (best-fit line) must contain a maximum number of datapoints.

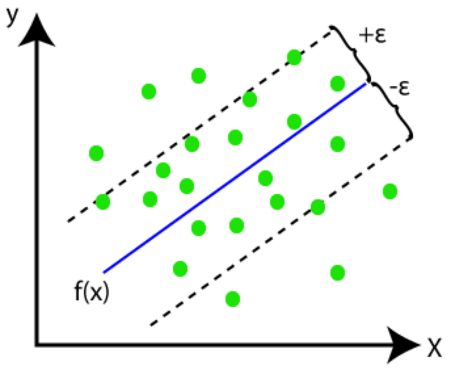


Figure 3.10: The blue line is called hyperplane, and the other two lines are known as boundary lines.[25]

Training the SVR means solving:

Minimize $\frac{1}{2}\|w\|^2$ subject to $|y_i - \langle w, x_i \rangle - b| \leq \varepsilon$

where x_i is a training sample with target value y_i, and w is the (not necessarily normalized) normal vector to the hyperplane. The inner product plus intercept $\langle w, x_i \rangle + b$ is the prediction for that sample, and ε is a free parameter that serves as a threshold: all predictions have to be within an ε range of the true predictions. Slack variables are usually added into the above to allow for errors and to allow approximation in the case the above problem is infeasible.

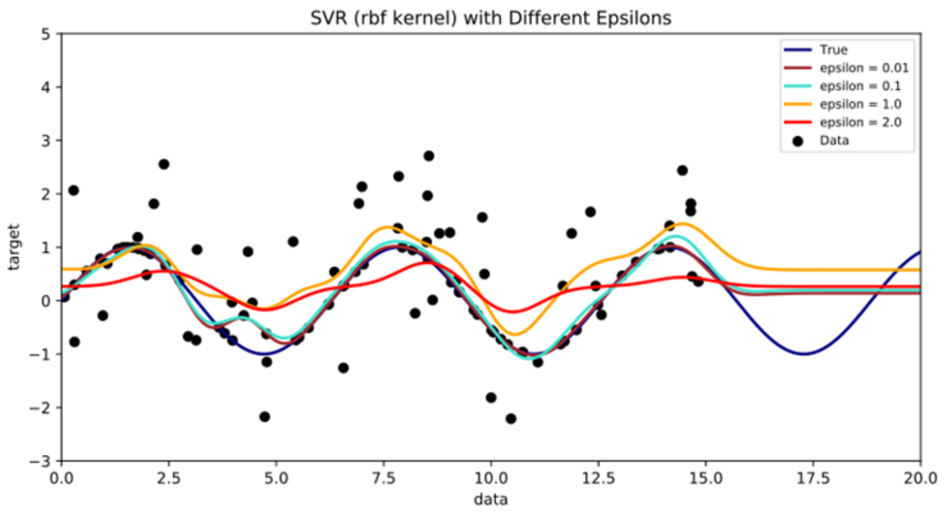


Figure 3.11: Support-vector regression (prediction) with different thresholds ε. As ε increases, the prediction becomes less sensitive to errors.[26]

[25] https://www.javatpoint.com/regression-analysis-in-machine-learning
[26] https://en.wikipedia.org/wiki/Support_vector_machine#Regression

K-nearest Neighbors Regression

In statistics, the k-nearest neighbors algorithm (k-NN) is a non-parametric supervised learning method, used for classification and regression [23]. In both cases, the input consists of the k closest training examples in a data set. In k-NN regression, the output is the property value for the object. This value is the average of the values of k nearest neighbors.

A useful technique can be to assign weights to the contributions of the neighbors, so that the nearer neighbors contribute more to the average than the more distant ones. For example, a common weighting scheme consists in giving each neighbor a weight of 1/d, where d is the distance to the neighbor. This scheme is a generalization of linear interpolation. The neighbors are taken from a set of objects for which the object property value (for k-NN regression) is known. This can be thought of as the training set for the algorithm, though no explicit training step is required.

A peculiarity of the k-NN algorithm is that it is sensitive to the local structure of the data.

In k-NN regression, the k-NN algorithm ιs used for estimating continuous variables. One such algorithm uses a weighted average of the k nearest neighbors, weighted by the inverse of their distance. This algorithm works as follows[27]:

1. Compute the Euclidean or Mahalanobis distance from the query example to the labeled examples.
2. Order the labeled examples by increasing distance.
3. Find a heuristically optimal number k of nearest neighbors, based on the root-mean-square error (RMSE). This is done using cross validation.
4. Calculate an inverse distance weighted average with the k-nearest multivariate neighbors.

Gaussian process regression (GPR)

Gaussian process regression (GPR) is a nonparametric, Bayesian approach to regression. Unlike many popular supervised machine learning algorithms that learn exact values for every parameter in a function, the Bayesian approach infers a probability distribution over all possible values. Let’s assume a linear function: y=wx+ϵ. How the Bayesian approach works is by specifying a prior distribution, p(w), on the parameter, w, and relocating probabilities based on evidence (i.e., observed data) using Bayes’ Rule[28]:

$$p(w|y,X) = \frac{p(y|X,w)p(w)}{p(y|X)}, \quad posterior = \frac{likelihood * prior}{marginal\ likelihood}$$

The updated distribution p(w|y, X), called the posterior distribution, thus incorporates information from both the prior distribution and the dataset. To get predictions at unseen points of interest, x*, the predictive distribution can be calculated by weighting all possible predictions by their calculated posterior distribution:

[27] https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm#k-NN_regression
[28] https://towardsdatascience.com/quick-start-to-gaussian-process-regression-36d838810319

$$p(f^*|x^*, y, X) = \int_w p(f^*|x^*, w)p(w|y.X)dw$$

Where we are calculating predictive distribution, f* is prediction label, x* is test observation.

The prior and likelihood is usually assumed to be Gaussian for the integration to be tractable. Using that assumption and solving for the predictive distribution, we get a Gaussian distribution, from which we can obtain a point prediction using its mean and an uncertainty quantification using its variance.

Gaussian process regression is nonparametric (i.e., not limited by a functional form), so rather than calculating the probability distribution of parameters of a specific function, GPR calculates the probability distribution over all admissible functions that fit the data. However, similar to the above, we typically specify a Gaussian process prior (on the function space), calculate the posterior using the training data, and compute the predictive posterior distribution on our points of interest.

Decision Tree Regression

Decision Tree is one of the most commonly used, practical approaches for supervised learning. It can be used to solve both Regression and Classification tasks with the latter being put more into practical application. Decision trees are among the most popular machine learning algorithms given their intelligibility and simplicity.

It is a tree-structured classifier with three types of nodes. The Root Node is the initial node which represents the entire sample and may get split further into further nodes. The Interior Nodes represent the features of a dataset and the branches represent the decision rules. Finally, the Leaf Nodes represent the outcome. This algorithm is especially useful for solving decision-related problems.

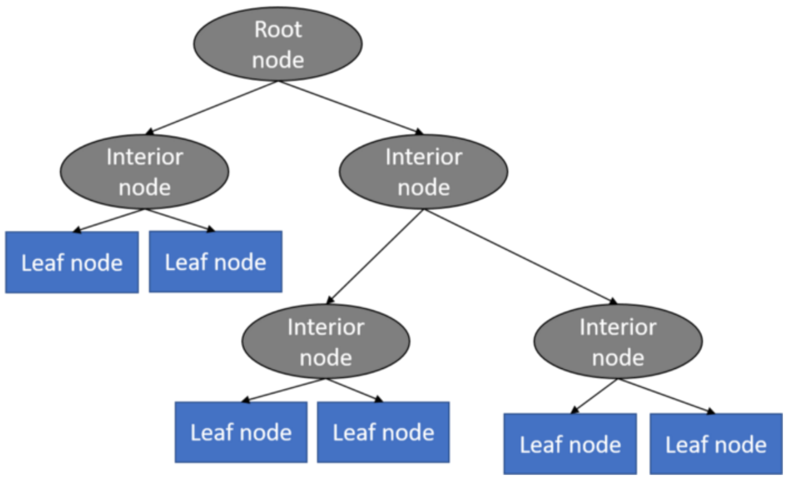


Figure 3.12: Decision Tree Algorithm[29]

[29] https://towardsdatascience.com/machine-learning-basics-decision-tree-regression-1d73ea003fda

With a particular data point, it is run completely through the entire tree by answering True/False questions till it reaches the leaf node. The final prediction is the average value of the dependent variable in that particular leaf node. Through multiple iterations, the Tree is able to predict a proper value for the data point.

Decision trees have an advantage that they are easy to understand, lesser data cleaning is required, non-linearity does not affect the model's performance and the number of hyper-parameters to be tuned is almost null. However, it may have an over-fitting problem, which can be resolved using the Random Forest algorithm.

Random Forest Regression

Random forest is one of the most powerful supervised learning algorithms which is capable of performing regression as well as classification tasks. Random Forest Regression is a supervised learning algorithm that uses an ensemble learning method for regression. Ensemble learning method is a technique that combines predictions from multiple machine learning algorithms to make a more accurate prediction than a single model. More specifically, Random Forest Regression combines multiple decision trees and predicts the final output based on the average of each tree output. The combined decision trees are called base models, and it can be represented more formally as:

$$g(x) = f_0(x) + f_1(x) + f_2(x) + \cdots$$

Random forest uses the Bootstrap Aggregating technique of ensemble learning, also called bagging, in which aggregated decision tree runs in parallel and do not interact with each other. With the help of Random Forest regression, we can prevent Overfitting in the model by creating random subsets of the dataset.

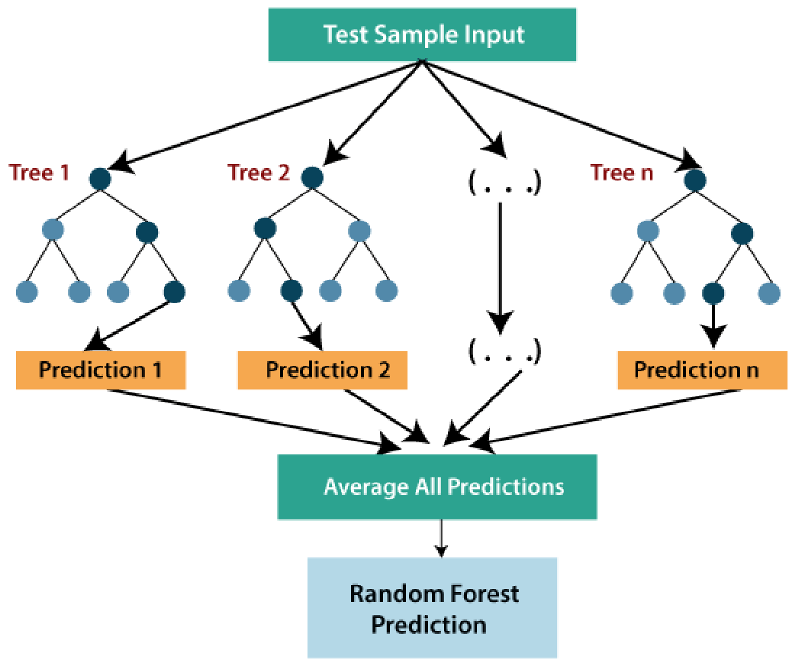


Figure 3.13: Random Forest Regression Algorithm[30]

Multi-layer Perceptron regression

Artificial neural networks (ANNs), usually simply called neural networks (NNs) or neural nets[31], are computing systems inspired by the biological neural networks that constitute animal brains.

An ANN is based on a collection of connected units or nodes called artificial neurons, which loosely model the neurons in a biological brain. Each connection, like the synapses in a biological brain, can transmit a signal to other neurons. An artificial neuron receives signals then processes them and can signal neurons connected to it. The "signal" at a connection is a real number, and the output of each neuron is computed by some non-linear function of the sum of its inputs. The connections are called edges. Neurons and edges typically have a weight that adjusts as learning proceeds. The weight increases or decreases the strength of the signal at a connection. Neurons may have a threshold such that a signal is sent only if the aggregate signal crosses that threshold.

Typically, neurons are aggregated into layers. Different layers may perform different transformations on their inputs. Signals travel from the first layer (the input layer) to the last layer (the output layer), possibly after traversing the layers multiple times.[32]

A multilayer perceptron (MLP) is a fully connected class of feedforward artificial neural network (ANN) that has 3 or more layers of perceptrons. These layers are- a single input layer, 1 or more hidden layers, and a single output layer of perceptrons. Multilayer perceptrons are sometimes colloquially referred to as "vanilla" neural networks, especially when they have a single hidden layer [24]. The data flows in a single direction, that is forward, from the input layers-> hidden layer(s) -> output layer.

Except for the input nodes, each node is a neuron that uses a nonlinear activation function. MLP utilizes a supervised learning technique called backpropagation for training [25]. Backpropagation is a technique where the multi-layer perceptron receives feedback on the error in its results and the MLP adjusts its weights accordingly to make more accurate predictions in the future. Its multiple layers and non-linear activation distinguish MLP from a linear perceptron. It can distinguish data that is not linearly separable. MLP is used in many machine learning techniques like classification and regression. They have been shown to give highly accurate results for classification problems in particular.

[30] https://discuss.boardinfinity.com/t/what-is-decision-tree-regression/4963
[31] https://news.mit.edu/2017/explained-neural-networks-deep-learning-0414
[32] https://en.wikipedia.org/wiki/Artificial_neural_network

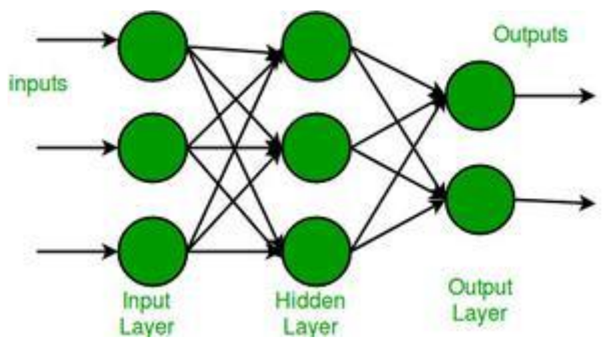


Figure 3.14: MLP layers[33]

XGBoost Regression

The XGBoost (eXtreme Gradient Boosting) algorithm is a highly optimized open-source implementation of the gradient boosting decision trees algorithm, created by Tianqi Chen, now with contributions from many developers. Shortly after its development and initial release, XGBoost became the go-to method and often the key component in winning solutions for a range of problems in machine learning competitions. Regression predictive modeling problems involve predicting a numerical value such as a dollar amount or a height. XGBoost can be used directly for regression predictive modeling.

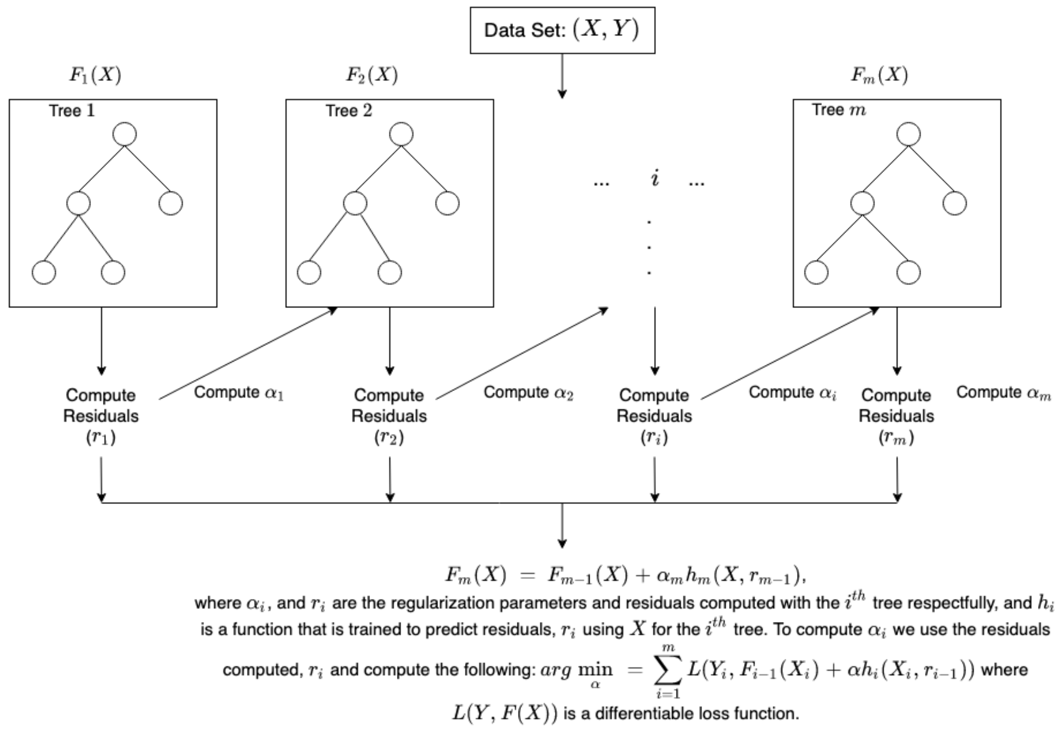
Gradient boosting refers to a class of ensemble machine learning algorithms that can be used for classification or regression predictive modeling problems. It is a supervised learning algorithm, which attempts to accurately predict a target variable by combining the estimates of a set of simpler, weaker models.

Ensembles are constructed from decision tree models. Trees are added one at a time to the ensemble and fit to correct the prediction errors made by prior models. This is a type of ensemble machine learning model referred to as boosting. Models are fit using any arbitrary differentiable loss function and gradient descent optimization algorithm. This gives the technique its name, "gradient boosting," as the loss gradient is minimized as the model is fit, much like a neural network.[34]

When using gradient boosting for regression, the weak learners are regression trees, and each regression tree maps an input data point to one of its leaves that contains a continuous score. XGBoost minimizes a regularized (L1 and L2) objective function that combines a convex loss function (based on the difference between the predicted and target outputs) and a penalty term for model complexity (in other words, the regression tree functions). The training proceeds iteratively, adding new trees that predict the residuals or errors of prior trees that are then combined with previous trees to make the final prediction. It's called gradient boosting because it uses a gradient descent algorithm to minimize the loss when adding new models.

Below is a brief illustration on how gradient tree boosting works.

[33] https://www.geeksforgeeks.org/difference-between-multilayer-perceptron-and-linear-regression/
[34] https://machinelearningmastery.com/xgboost-for-regression/



$$F_m(X) \; = \; F_{m-1}(X) + \alpha_m h_m(X, r_{m-1}),$$

where α_i, and r_i are the regularization parameters and residuals computed with the i^{th} tree respectfully, and h_i is a function that is trained to predict residuals, r_i using X for the i^{th} tree. To compute α_i we use the residuals computed, r_i and compute the following: $arg \min_{\alpha} \; = \sum_{i=1}^{m} L(Y_i, F_{i-1}(X_i) + \alpha h_i(X_i, r_{i-1}))$ where $L(Y, F(X))$ is a differentiable loss function.

Figure 3.15: Gradient Boosting algorithm illustration[35]

XGBoost is designed to be both computationally efficient (e.g., fast to execute) and highly effective, perhaps more effective than other open-source implementations. The two main reasons to use it are execution speed and model performance. XGBoost dominates structured or tabular datasets on classification and regression predictive modeling problems. The evidence is that it is the go-to algorithm for competition winners on the Kaggle competitive data science platform.

[35] https://docs.aws.amazon.com/sagemaker/latest/dg/xgboost-HowItWorks.html

Chapter 4

Characterization

In this chapter, we describe the cluster we have created for our experiments, and we examine the behavior of the benchmarks from the MLPerf Inference Benchmark Suite by deploying them to run in the cluster in a SingleStream or MultiStream scenario, with or without external sources of interference.

4.1 Experimental Infrastructure

4.1.1 System Setup

For the experiments of our thesis, we have deployed 2 virtual machines (VMs) on top of the physical machines with different configurations, serving as the nodes of our cluster (1 master node and 1 worker node). The master node has 4 processing cores and 8GB of RAM, while the worker node has 8 processing cores and 16GB of RAM. We used KVM as our hypervisor. To simulate a cloud environment, all the referenced workloads running on the cluster have been containerized, utilizing the Docker platform.

Virtual Machines		
VM-name	**Cores**	**RAM (GB)**
master	4	8
worker	8	16

Table 4.1: Virtual Machines Characteristics

The combination of VMs with containers is currently the common way of deploying cloud clusters at scale, since it establishes the perfect catalyst for reliability and robustness. On top of the VMs, we have deployed Kubernetes as our container orchestrator, one of the most popular and most used platforms nowadays.

4.1.2 MLPerf Inference Benchmarks for Image Classification and Object Detection Tasks

From the Kubernetes perspective the workload consists of a set of Pods, controlled by Jobs that start the previously described MLPerf Inference Benchmarks. We created a container image of the MLPerf Inference benchmark suite that, given the appropriate parameters (model, backend, scenario, time etc.), starts the different benchmarks. *We will use the MLPerf Inference benchmarks as the Inference Engines for our experiments and the design of our schedulers.*

4.1.3 iBench

In our thesis we use the following micro-benchmarks from the iBench suite[36]:

- CPU
- L2 cache
- L3 cache
- Memory Bandwidth stress
- Memory Capacity stress

We deploy the iBench benchmarks as jobs in the Kubernetes cluster. The duration of each iBench job is set at a constant value of 3600s in their deployment yaml file. This is to ensure that the impact each iBench micro-benchmark has on a shared resource is increasing at a low enough pace to practically consider each iBench job to create a small amount of interference of constant intensity throughout our experiments. Consequently, we control the intensity of the stress, the iBench SoIs create at the shared resources of our cluster, by adjusting the number of concurrent iBench workloads that are pressuring each resource accordingly.

4.2 SingleStream Benchmark Scenario

In this first series of experiments, we will be deploying the models from the MLPerf Inference Benchmark suite, presented in chapter 3, as Kubernetes pods, using the SingleStream scenario. We test the performance of the benchmarks with and without the presence of external sources of interference, as well as with CPU limitations imposed on them by the Kubernetes system.

4.2.1 With interference

We start our experiments by deploying the models from the MLPerf Inference Benchmarks as Kubernetes pods, using the SingleStream scenario. For each model running, we measure the queries per second (QPS) it achieves, as a benchmark performance indicator, while varying the intensity of a specific iBench micro-benchmark (pressuring a specific shared resource). The intensity of the iBench interference is controlled accordingly by the number of “iBench” jobs that are deployed as Kubernetes pods, concurrently with the running MLPerf model.

[36] https://github.com/stanford-mast/iBench

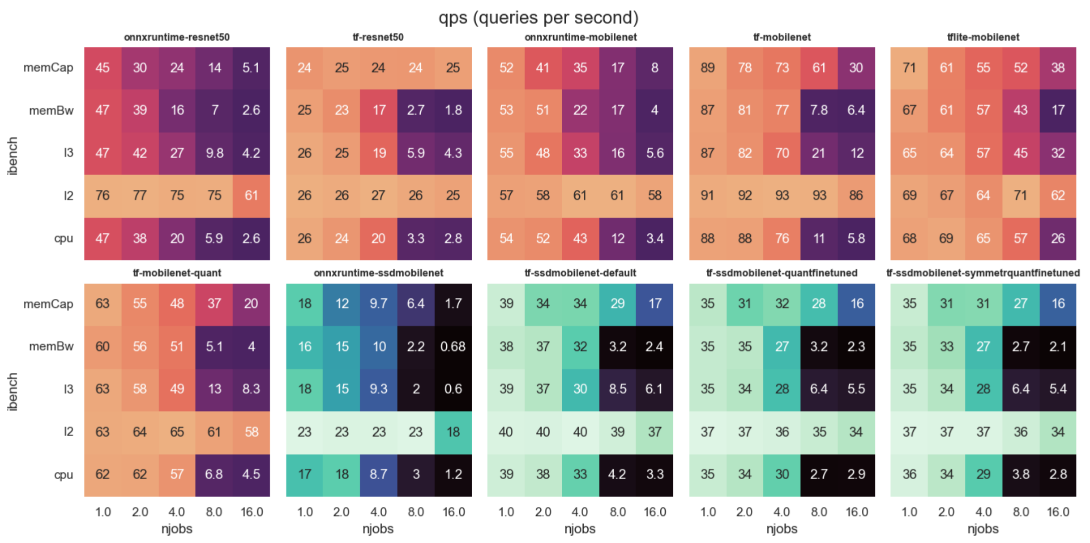


Figure 4.1: Heatmap: QPS measurement (colored squares) of the MLPerf Inference Benchmarks for various “iBench” sources of interference at different intensities (njobs). The first six benchmarks are running an image classification task, while the last four benchmarks are running an object detection task, as designated by their different color palette.

We measure the achieved QPS of each MLPerf Inference model, at the presence of pressure on a specific shared resource at each time (cpu, l2 cache, l3 cache, memory bandwidth and memory capacity) induced by the iBench micro-benchmarks. For each resource in contention, the amount of pressure it receives increases accordingly as the number of concurrent “iBench” jobs increases from 1 to 16. The measured QPS is represented in a colored heatmap in the above figure, where the darker color each square has, the more severe drop in the expected QPS for the MLPerf Inference benchmark we detect. Similarly, the lighter the color of the square, the higher QPS the model achieves under the stressful circumstances.

There are some interesting points we gather from the above figure as far as each MLPerf Inference model’s behavior under pressure is concerned.

Firstly, we see that in general, the more “iBench” jobs we have for each shared resource, meaning the more intensely pressured the resource is, the more the QPS of the model drops to a lower value. However, for the contention of the L2 cache, it is shown that, with up to 8 concurrent “iBench” jobs pressuring it, there is no quantifiable effect on the achieved QPS of all the models. Even at 16 “iBench” jobs, the drop in QPS is minuscule, leading us to the conclusion that the L2 cache shared resource is of no particular significance to our MLPerf Inference benchmarks, especially at lower intensities of contention. That means that our models can withstand excessive amounts of pressure on the L2 cache before their performance begins to decrease.

Secondly, we see that while all models are susceptible to performance degradation as the stress on memory capacity, placed by the “iBench” jobs, increases, this is not true for the resnet50 model with tensorflow backend. In contrast to even the same model with the onnxruntime backend, the tf-resnet50 model maintains the same QPS under any memory capacity stress, no matter how intense. This could make tf-resnet50 a great model candidate, when performance stability under restricted memory capacity is required.

Thirdly, as far as measured performance goes, it is obvious from the figure that the mobilenet model with the tensorflow backend manages to score higher in the image classification task than any other model-backend combination, and it should be the model of choice for environments with resources that are not being heavily pressured. Tf-mobilenet holds the highest QPS scores, and the best performance output, under all different shared resources contentions, with resource stress intensity up to 4 concurrent "iBench" jobs. For higher resource stress intensities (8 or more concurrent "iBench" jobs), we observe that the mobilenet model with the tflite backend appears to be the more resilient. Tflite-mobilenet seems to be the best able to hold its QPS output higher than the other models during high shared resource contention, as we notice the lowest benchmark performance percentage drop as "iBench" jobs multiply. Furthermore, tflite-mobilenet manages to achieve high QPS values at even 16 concurrent "iBench" jobs, making it a fine choice at environments where sources of interference are plentiful.

Another interesting observation from the figure is that, for the image classification task, the resnet50 model performs better overall with the onnxruntime backend than with the tensorflow backend. In contrast, mobilenet shows the best overall performance with the tensorflow backend, a little worse performance with the tflite backend and the worst performance comperatively with the onnxruntime backend.

Lastly, at the object detection task of the MLPerf Inference benchmarks, we easily conclude that all the ssd-mobilenet models with the tensorflow backend achieve higher overall QPS, with little variance between them, than the ssd-mobilent model does with the onnxruntime backend.

4.2.2 Without interference

The last two observations can better be visualized by the following figure. In this experiment, we deploy again the models from the MLPerf Inference Benchmarks as Kubernetes pods, using the SingleStream scenario. However, here, we simplify our approach by not adding any source of interference to our cluster and thus measuring each model's best possible performance, which is quantified by the highest score of queries per second (QPS) each model can achieve.

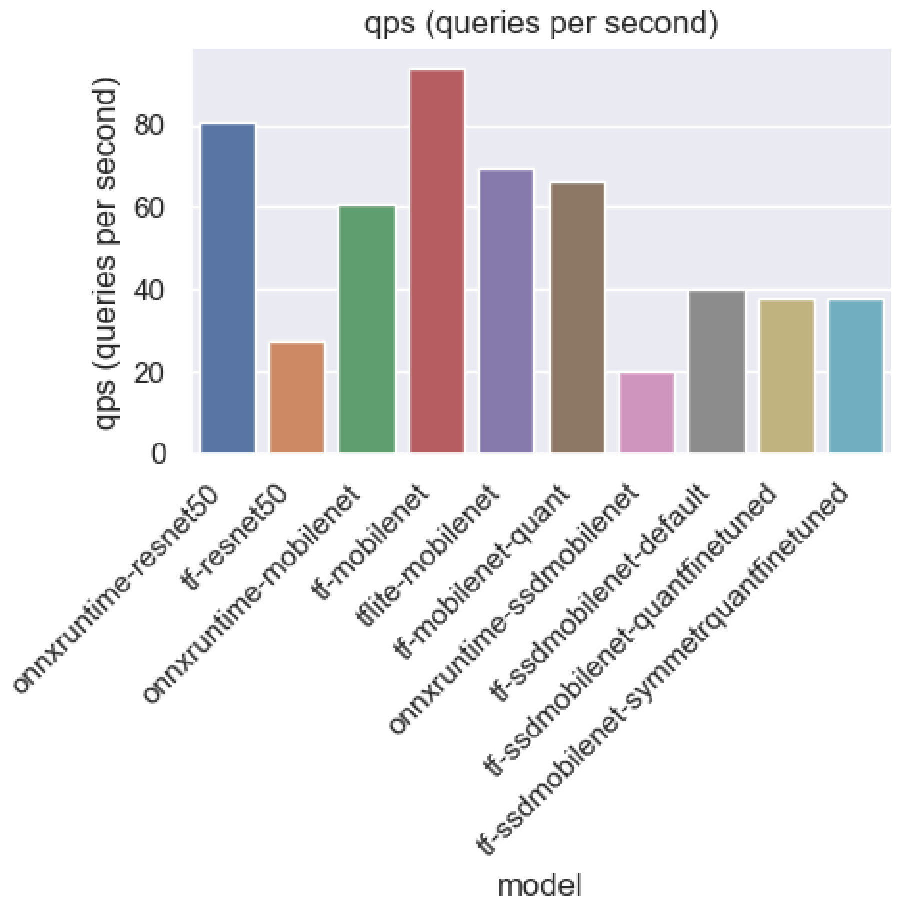


Figure 4.2: Performance measurement (QPS) of the MLPerf Inference Benchmarks in SingleStream scenario. The first six benchmarks are running an image classification task, while the last four benchmarks are running an object detection task.

From this figure, we confirm our previous observations, having the mobilenet model performing better with the tensorflow backend, while the resnet50 model scoring higher with the onnxruntime backend in the image classification task. It is also clear that all the ssd-mobilenet models achieve a higher QPS score with the tensorflow backend than with the onnxruntime backend in the object detection task.

An interesting conclusion we can easily derive from this figure however, is that, in the image classification task, the highest overall inference performance is attained by the tf-mobilenet at around 93 QPS, with the second best, the onnxruntime-resnet50, scoring quite a bit lower at around 80 QPS, a more than 13% reduction. On the other hand, in the object detection task, the highest performer, the tf-ssd-mobilenet-default, scores at around 40 QPS, less than half the best score in the image classification task. Despite that, the performance difference from the second and third best, the other ssd-mobilenet models with the tensorflow backend, is not as big, being less than 6%.

4.2.3 CPU Limits

For our next experiment, we run only the MLPerf Inference benchmarks that perform an image classification task, using the SingleStream scenario again. The models are deployed over the Kubernetes platform as Pods. Once more, no sources of external interference are added to the cluster. For each model running, we measure the queries per second (QPS) it achieves, as a benchmark performance indicator, while imposing varying levels of cpu limititations on them. The cpu limits are implemented as a resource limit for the benchmark containers. The kubelet then enforces those limits so that the running container is not allowed to use more of that resource than the set limit.[37] We also compare the benchmarks' inference accuracy, given by table 3.1, as is illustrated in the figure below.

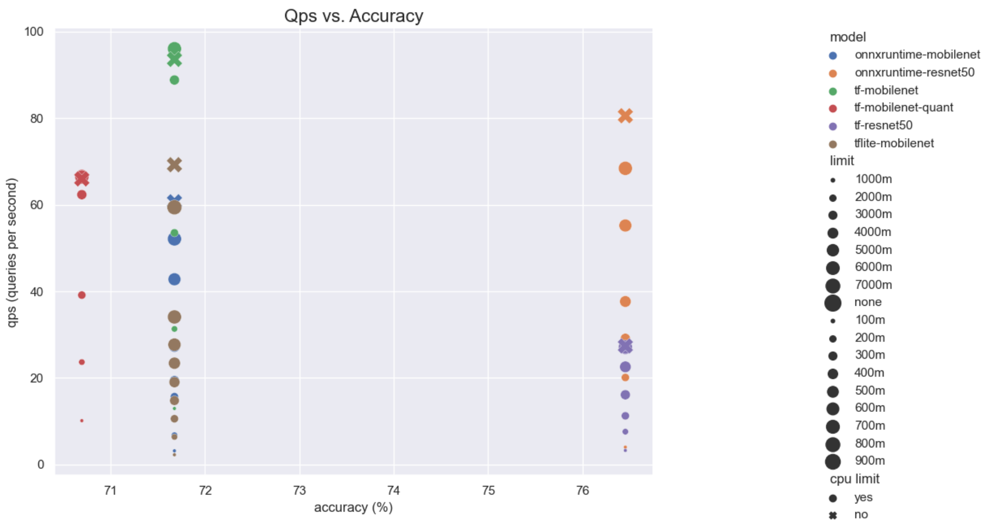


Figure 4.3: QPS vs. Accuracy of the MLPerf Inference benchmarks (each benchmark presented with its own color) that perform an image classification task, with various imposed CPU limits.

We measure the achieved QPS of each MLPerf Inference model, while hindering in a controlled way their performance by placing cpu limitations varying from using small fractions of the available cpu power to not limiting the cpu usage at all. The larger the colored dots on the figure, the smaller the imposed limit, meaning the more available cpu the benchmark has for its own. The x marks on the figure indicate that, for those measurements, there is no limit on the cpu usage for the benchmarks.

It is also worth noting that, although all the models we tested make use of all the available cpu cores, filling them at different amounts depending on the cpu limit percentage (1000m-7000m), the mobilenet model with the tflite backend only utilizes one core. For this reason, the cpu limits imposed to tflite-mobilenet are tailored to split only a single core into smaller fractions (100m-900m), so we can measure the limitations' effect in comparison to that on multiple cores for the other models.

From the figure above, as far as the MLPerf Inference benchmarks' accuracy is concerned, it is clear that the resnet50 model sits further on the accuracy line, both with the onnxruntime and tensorflow backend, making it the most accurate inference model between them. Similarly, the least accurate model seems to be the quantized mobilenet with the tensorflow backend.

[37] https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/

As far as the MLPerf Inference benchmarks' performace goes, we see that the highest QPS scores, when cpu limits become smaller, are achieved by the mobilenet model with the tensorflow backend, albeit not the best model accurate-wise. The mobilenet model with the onnxruntime or the tflite backend scores lower on the performance metric, while having the same accuracy percentage. On the other hand, between the two most accurate benchmarks, the resnet50 model with the onnxruntime backend considerably outperforms its tensorflow backend counterpart.

Another interesting observation from the figure is that the models with the onnxruntime or tflite backend have a clear performance increase between the runs with the lowest cpu limitations and with no limitations. In other words, even if a small fraction of the cpu is not available, their QPS score drops. On the contrary, all the models with the tensorflow backend score close to their maximum QPS value when cpu limitations become small, meaning they do not decrease their performace when a small fraction of the cpu is unavailable.

4.3 MultiStream Benchmark Scenario

For the next series of experiments, we will be deploying the models from the MLPerf Inference Benchmark suite, presented in chapter 3, as Kubernetes pods, using the MultiStream scenario. We exclude the mobilenet model with the tflite backend, since it does not run on a MultiStream benchmark scenario. For each model running, we will measure the queries per second (QPS) it achieves, as a benchmark performance indicator, under different interference conditions and adjustments of some environment variables.

4.3.1 Onnxruntime framework - without interference

In the first experiment, we deploy only the MLPerf Inference Benchmarks with the onnxruntime backend. We compare the QPS score these models achieve for a combination of different values of the OMP_NUM_THREADS variable (provided by the OpenMP API specification for parallel programming) and the --threads option (provided by the MLPerf Inference benchmark suite). The --threads option of the MLPerf Inference benchmark suite defines the number of worker threads to use, while the OMP_NUM_THREADS environment variable sets the number of threads to use for parallel regions[38]. In other words, the --threads option instructs the number of concurrent "inference engine instances" or "inference workers" each MLPerf Inference Benchmark deploys, while the OMP_NUM_THREADS variable sets the number of threads per concurrent "inference engine instance". No source of external interference is added to our cluster.

[38] https://www.openmp.org/spec-html/5.0/openmpse50.html

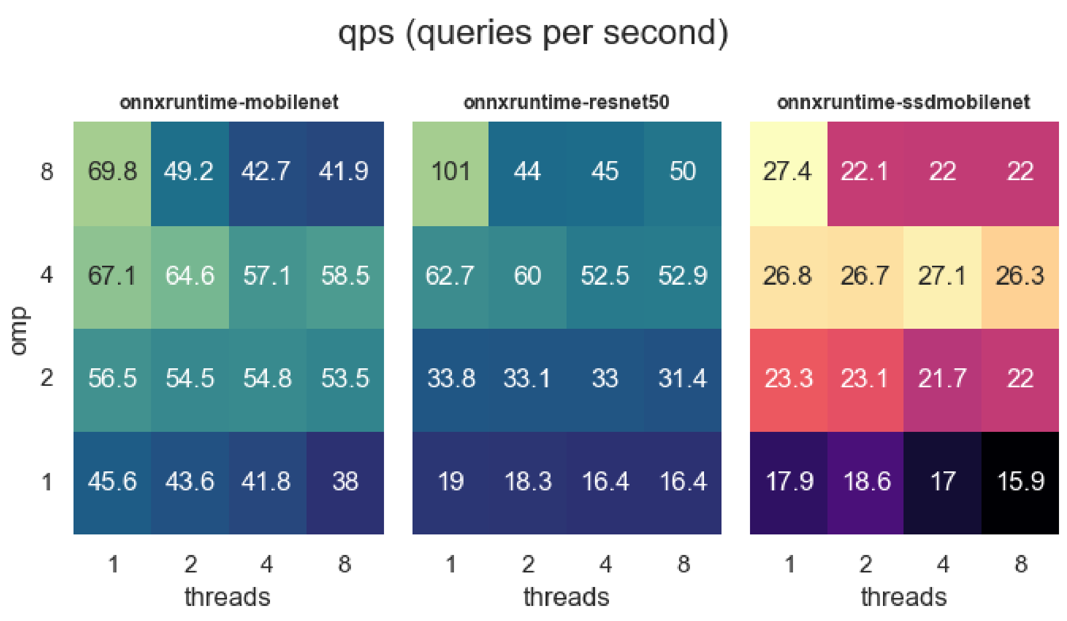


Figure 4.4: Heatmap: QPS measurement (colored squares) of the MLPerf Inference Benchmarks with the onnxruntime backend, for various combinations of the OMP_NUM_THREADS variable and the --threads option. The first two benchmarks are running an image classification task, while the third benchmark is running an object detection task, as designated by their different color palette.

We measure the QPS score of the MLPerf Inference Benchmarks with the onnxruntime backend at each combination of omp (OMP_NUM_THREADS) and threads with values of 1, 2, 4 and 8.

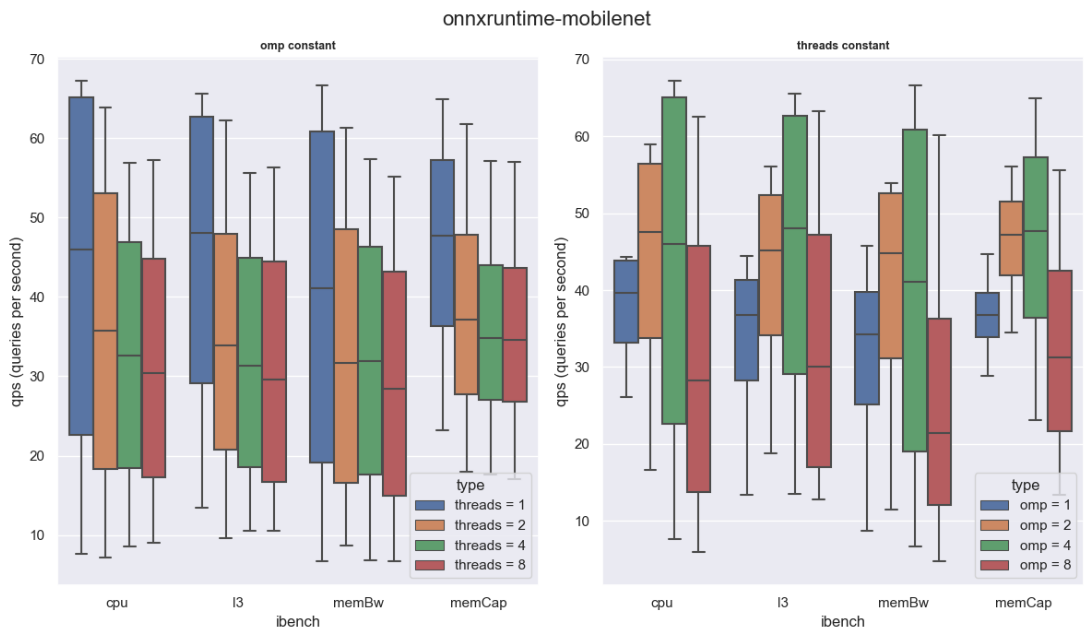
From the figure, we see a trend in all three of our models, that we achieve better overall performance at omp values of 2 or more, regardless of the amount of threads. We also notice that the best performance is attained when the omp variable is at 4 and threads at 1 or 2, or when we have the omp variable at 8 and threads at 1.

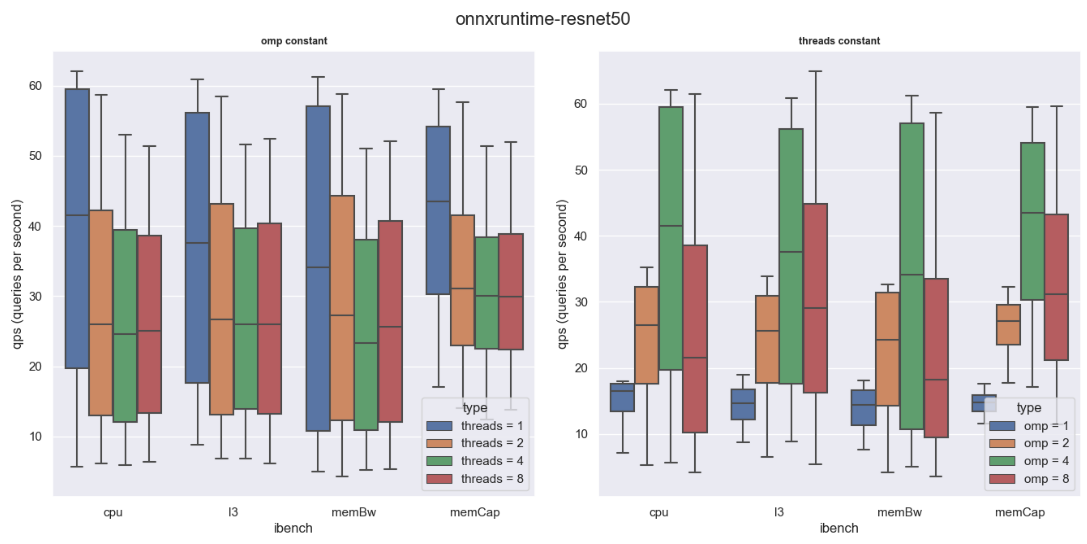
More specifically, for the image classification task, we observe that the mobilenet model manages to score higher than the resnet50 model in almost all the combinations, making it the best performing model. The onnxruntime-mobilenet also appears to be the most stable one performance-wise across the changing variables, since we notice a smaller QPS drop at an omp value of 1, as opposed to the respective behavior of resnet50 at omp 1. On the other hand, the resnet50 model achieves an outstanding score at omp 8 and threads 1, but its performance on the other combinations of the variables drops significantly.

As far as the object detection task is concerned, we observe almost no variance of the QPS values at the highest performance configurations of omp 4, or omp 8 and threads 1, unlike the onnxruntime models tasked with image classification, whose QPS score on these configurations varies more.

4.3.2 Onnxruntime framework - with interference

Next, we continue our testing of the MLPerf Inference Benchmarks with the onnxruntime backend, only this time we are bringing external interference into the Kubernetes cluster, induced by the iBench micro-benchmarks. For this experiment, we are measuring the QPS each model achieves while the cluster is being pressured at a specific shared resource at a time, as well as the variance in the QPS attained value under the contention of these resources, at various intensities, by the iBench jobs. Then, for each onnxruntime model, we compare the QPS score and its variance under the contention of different resources, by keeping one environment variable constant and changing the value of the other (the two variables we control are: the OMP_NUM_THREADS variable and the --threads option). As a result, we generate two subplots for each model, one for each variable being kept constant, to see the effect that the other free-to-change variable has on the QPS and its variance, under interference.





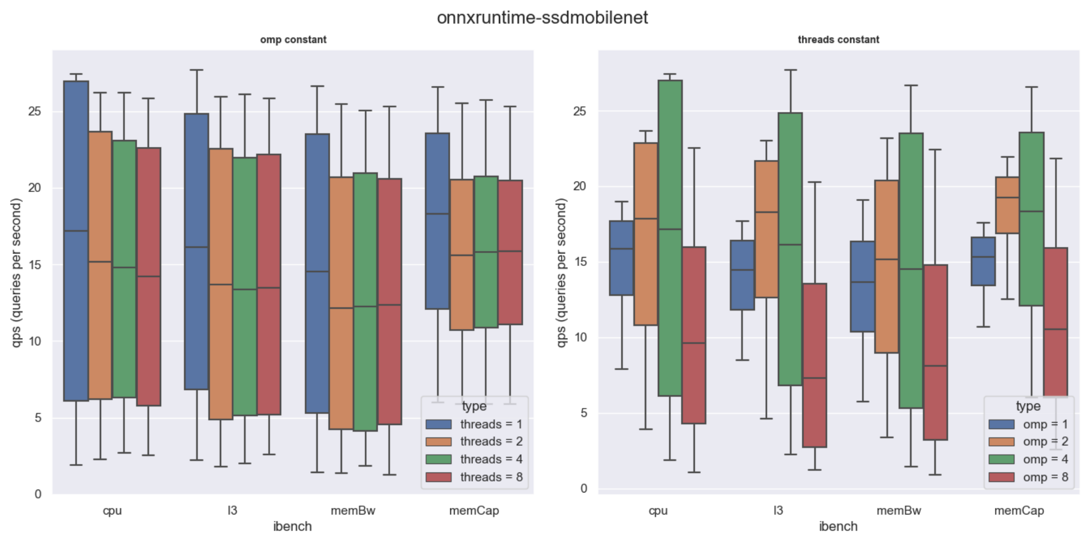


Figure 4.5: Boxplots: QPS measurement of the MLPerf Inference Benchmarks with the onnxruntime backend, for various “iBench” sources of interference at different intensities. In each MLPerf Inference Benchmark’s plot, there are two subplots, one for each constant environment variable. On the left subplot, the OMP_NUM_THREADS variable is kept constant, while on the right subplot the --threads option is constant. The achieved QPS at each iBench micro-benchmark is measured for various values of the free-to-change variable at each subplot, as designated by the different color (type) of the boxplots. The first two benchmarks are running an image classification task, while the third benchmark is running an object detection task.

For each MLPerf Inference model, we deploy iBench jobs pressuring a specific shared resource at a time (cpu, l3 cache, memory bandwidth and memory capacity) at different intensities, with the number of concurrent “iBench” jobs pressuring the same resource varying from 1 to 16. This creates a variability in the QPS measurement for each pressured resource, as observed in the above figure (indicated by the width of the boxplot). Furthermore, by holding one variable constant (either the OMP_NUM_THREADS variable or the --threads option) we compare how the QPS and its variance changes for values of 1, 2, 4 and 8 of the other variable, at each resource in contention.

From the above figure, we can derive some interesting points about the behavior of the three MLPerf Inference models with the onnxruntime backend under resource pressure and different variables combination.

To begin with, we focus on the subplots where we keep the omp (OMP_NUM_THREADS) environment variable constant. We observe for all three of the MLPerf Inference models, that we get consistently significantly higher QPS score for the --threads option at value 1 (blue box) than we do with higher values. The performance we get for the values 2, 4 and 8 of the --threads option is roughly similar, with the value of 2 having a small edge over the others, especially for the models of the image classification task.

We also see in these subplots, that there is a large amount of variability in the performance of the models, measured under different sources of interference at different intensities. The highest performance variability is seen again for the --threads option at value 1 (blue box). It is worth noting that for the higher values of --threads (higher than 1), this QPS variance is, to a large extent, independent of the value of the --threads option. Between the different sources of interference though, when the shared resource under stress is the memory capacity of the cluster, that performance variability seems to reduce comparatively for all the –threads values.

Bringing our attention to the subplots where we keep the --threads option constant, we see that the performance variability at each resource in contention, differs considerably in comparison, as the omp environment variable changes. In particular, we observe little variance in QPS when the omp variable is at 1 (blue box) for all three of the MLPerf Inference models. Noticeably, the performace variability increases when the omp variable has a value of 2, it increases even more at omp value of 8 and we get the highest variability at omp value of 4 (green box). These results suggest that our models are more robust performance-wise with an omp value of 1, albeit not achieving the highest QPS score at that variable value.

For discerning the best omp variable value for the highest QPS score, we have to differentiate between our MLPerf Inference models. For the resnet50 model with the onnxruntime backend, it is clear that it performs remarkably better with an omp value of 4. Even though at that value of the omp variable the model has the largest performance variability, the median QPS value of the corresponding box (green) in the figure is a lot higher than the rest of the boxes with the other omp values. It is also worth mentioning that the onnxruntime-resnet50 performs the worst at the omp value of 1, although it has the most robust performance, with the least variability at that omp value.

On the other hand, the mobilenet and ssd-mobilenet models with the onnxruntime backend show the best performance at omp values of 2 and 4. When the omp variable is at 2 (yellow box), we get a slightly higher median QPS value than what we have at omp value of 4, for most of the pressured resources. However, at omp value of 4 (green box), due to the higher performance variability, the QPS score can reach values higher than the ones achievable at omp value of 2, for the lowest amounts of pressure on each shared resource. Interestingly, these two models seem to perform the worst at an omp value of 8 (red box).

4.3.3 Tensorflow framework - without interference

In the following experiment, we deploy only the MLPerf Inference Benchmarks with the tensorflow backend. We measure the QPS (queries per second) these models achieve for various values of the INTRA_OP_PARALLELISM_THREADS option provided by the Tensorflow platform. The execution of an individual operation (for some op types) can be parallelized on a pool of intra_op_parallelism_threads, with a zero-value meaning the system picks an appropriate value.[39] In other words, the intra_op_parallelism_threads option, sets the number of threads used within an individual operation (like matrix multiplication and reductions) for parallelism[40]. The MLPerf Inference Benchmarks run operations that can utilize parallel threads for speed ups. The effect that the intra_op_parallelism_threads option (referred from now on as “tf_intra” for short) has on the performance of the MLPerf Inference models can be seen on the following figure. No source of external interference is added to our cluster for this experiment.

[39] https://github.com/tensorflow/tensorflow/blob/26b4dfa65d360f2793ad75083c797d57f8661b93/tensorflow/core/protobuf/config.proto#L165
[40] https://www.tensorflow.org/api_docs/python/tf/config/threading/set_intra_op_parallelism_threads

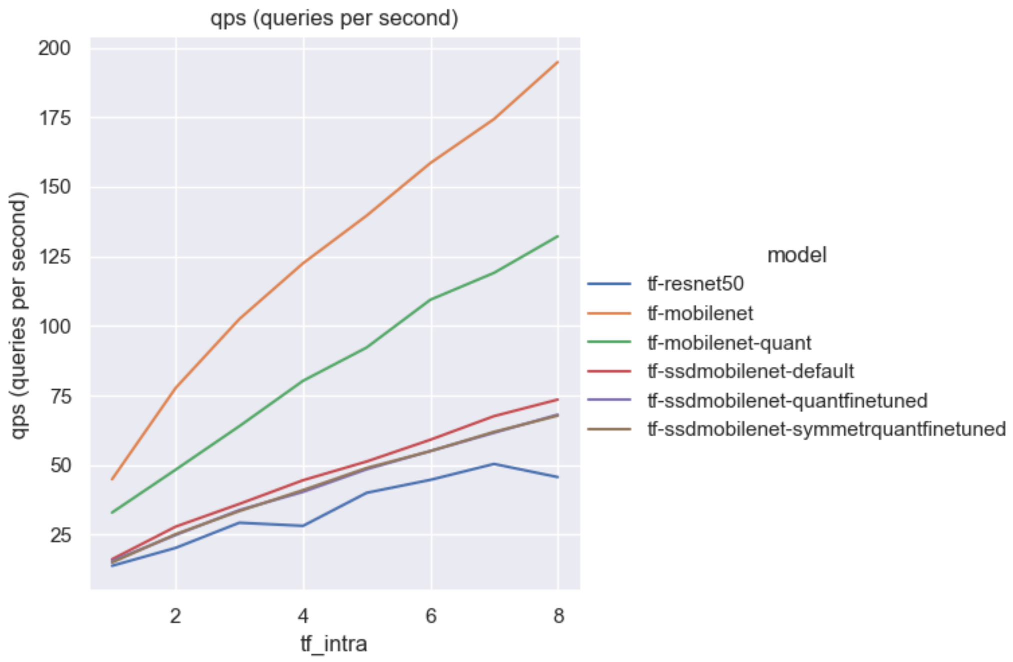


Figure 4.6: QPS measurement of the MLPerf Inference Benchmarks with the tensorflow backend, for various values of the INTRA_OP_PARALLELISM_THREADS option (tf_intra). The first three benchmarks, in the legend with the colours on the right, are running an image classification task, while the last three benchmarks are running an object detection task.

We measure the QPS score of the MLPerf Inference Benchmarks with the tensorflow backend at tf_intra values ranging from 1 to 8.

At first glance of the figure, we can readily conclude that the higher the tf_intra value is, meaning the greater the amount of parallel threads that are utilized by the MLPerf Inference models is, the better the performance of the models is, in a rather linear way. So, at the tf_intra values of 7 or 8 we get the highest QPS score for each model.

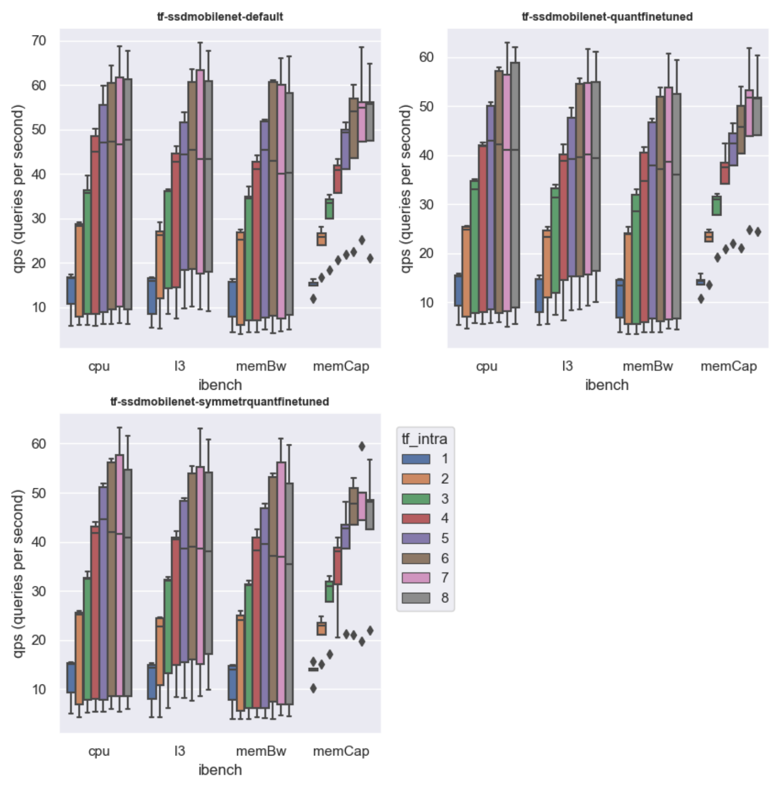
An interesting observation from the figure is that, even though all models start at a low QPS score for tf_intra at 1, the rate at which their performance improves, with the increase of the tf_intra option, differs considerably from model to model. Starting with the image classification task, we notice that the tf-mobilenet model (orange line in the figure) has, by far, the steepest angle of the performance increase, while also scoring the highest QPS at each tf_intra value. Performing a little worse with a lower angle, but still standing significantly above the rest of the competition, is the quantized tf-mobilenet model (green line), which also manages to outperform the rest of the models (except the aforementioned tf-mobilenet) at each tf_intra value. The four remaining models have a rather similar, lower angle of performance increase. Most noticeably, the tf-resnet50 model (blue line) presents the shallowest angle between them, while scoring the lowest QPS across all models at each tf_intra value.

It is also worth noting that the three models on the tensorflow backend performing the image classification task have largely different performance increase angles, whereas the three tensorflow models performing the object detection task have very similar low angles. In fact, the default tf-ssd-mobilenet model (red line) has a small edge on performance at each tf_intra value, as seen in the figure. The quantized, fine-tuned tf-ssd-mobilenet model and the symmetrically quantized, fine-tuned tf-ssd-mobilenet model have almost the same performance angle and QPS scores, and thus are indiscernible from one another in the figure (purple and brown lines accordingly), since their lines intersect at each and every tf_intra value.

4.3.4 Tensorflow framework - with interference

The next experiment consists of deploying the MLPerf Inference Benchmarks with the tensorflow backend, together with external sources of interference to measure the QPS they achieve, as well as the performance variability the models have, as the intensity of the interference varies. For each MLPerf Inference model and each shared resource in pressure, we measure the performance of the models for various values of the INTRA_OP_PARALLELISM_THREADS option provided by the Tensorflow platform. The resulting figure is the following.

Qps (queries per second) (with interference)



Qps (queries per second) (with interference)

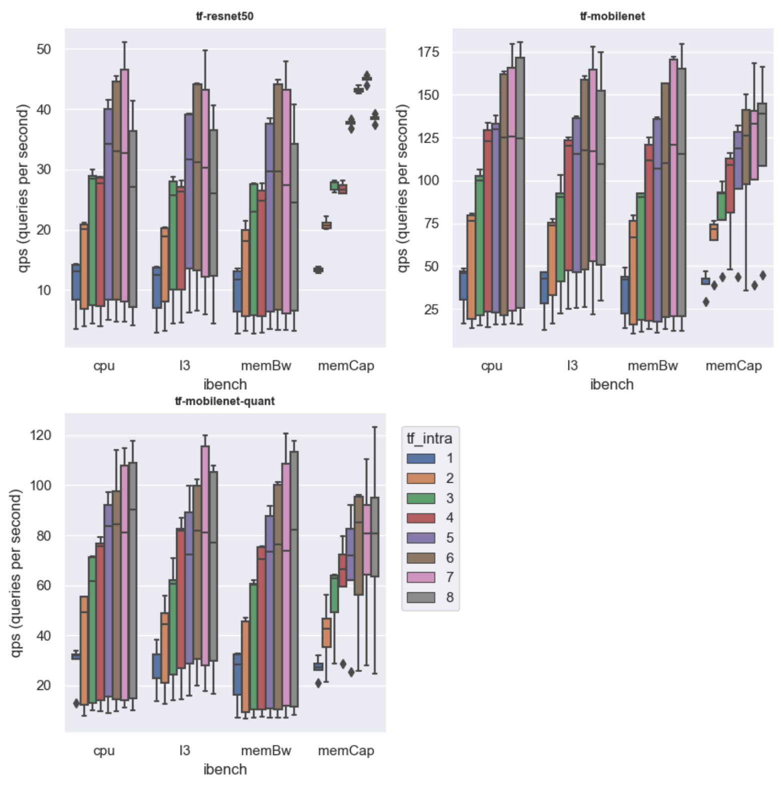


Figure 4.7: Boxplot: QPS measurement of the MLPerf Inference Benchmarks with the tensorflow backend, for various “iBench” sources of interference at different intensities. The achieved QPS at each iBench micro-benchmark is measured for various values of the INTRA_OP_PARALLELISM_THREADS option (tf_intra), as designated by the different color of the boxes. The first three benchmarks are running an image classification task, while the last three benchmarks are running an object detection task.

For each MLPerf Inference model, we deploy iBench jobs pressuring a specific shared resource at a time (cpu, l3 cache, memory bandwidth and memory capacity) at different intensities, with the number of concurrent “iBench” jobs pressuring the same resource varying from 1 to 16. This creates a variability in the QPS measurement for each pressured resource, as observed in the above figure (indicated by the width of the boxplot). Furthermore, we compare how the QPS and its variance changes for values ranging from 1 to 8 of the tf_intra option, at each resource in contention.

Starting off, we see a similar upward trend in performance across all the MLPerf Inference Benchmarks with the tensorflow backend, as the value of tf_intra increases. The median QPS value appears to be the lowest at tf_intra 1 (blue box) for all the tensorflow models. Despite that, at tf_intra 1 we notice the lowest performance variability (width of the box) on all of the models, on each different source of interference of varying intensity, meaning that the models are more robust in their performance at this configuration, although their QPS scores are typically very low. The highest QPS scores are generally noticed for tf_intra values of 5 and above. Due to the large amount of observed performance variability and the perceived small difference between the median values of QPS achieved by the models at the highest tf_intra values though, it is difficult to identify the exact tf_intra value for each model’s highest performance.

Another interesting observation from the above figure, that is true for all the tensorflow models, is that there is high performance variability while there is pressure of varying intensity, either in the cpu, the l3 cache or the memory bandwidth of the cluster. However, as the stress intensity in the memory capacity shared resource fluctuates, the performance of the models remains fairly robust with little diversion from the median value of the achieved QPS (except, maybe, for the highest level of stress intensity on the shared resource, as shown by the outlier values of QPS at the bottom of each box at the memory capacity section in the figure). It is also worth mentioning that the greatest performance sturdiness, during the varying amounts of pressure on the memory capacity of the system, is achieved by the resnet50 model with the tensorflow backend. Indeed, looking at the figure, we see there is almost no discernable variance in the QPS the tf-resnet50 model achieves at the memory capacity pressure setting.

Chapter 5

Scheduler

In this chapter, we present the training of some Machine Learning Regression models, by gathering a dataset of various system metrics. Furthermore, we explore what is the best ML model to use for QPS predictions in our custom scheduler and tune its hyperparameters to further improve its accuracy. Next, we analyze the algorithm that our custom, model-specific Inference Engine scheduling mechanism uses to serve MLPerf Inference benchmarks to the cluster. Finally, we introduce a model-less approach to the inference serving of our custom scheduler.

5.1 Offline: Training of the Machine Learning Regression Models

We want our proposed Inference Engine scheduling mechanism to be able to deploy the appropriate MLPerf Inference benchmark as a Kubernetes pod, taking into consideration the current usage level of various system resources, as well as the requests of the user.

In order to determine what MLPerf Inference benchmark, and with what configuration, is suitable for deployment at each time, we turn to the help of a set of Machine Learning Regression Models that, adequately trained, will be able to predict the potential performance of the benchmarks under the interference conditions that exist at the deployment time.

The training of the Machine Learning Regression Models was done in advance, by collecting a set of various system metrics at different interference scenarios and measuring the corresponding performance of the MLPerf Inference benchmarks. With the resulting training dataset, we trained and evaluated the ML Regression models and we picked the most accurate one, in predicting the QPS scores of the benchmarks, to use in our custom scheduler. Finally, we tuned its hyperparameters, for each benchmark, to optimize its accuracy further.

The following diagram follows the aforementioned steps of the offline training and ML model selection for our custom scheduler.

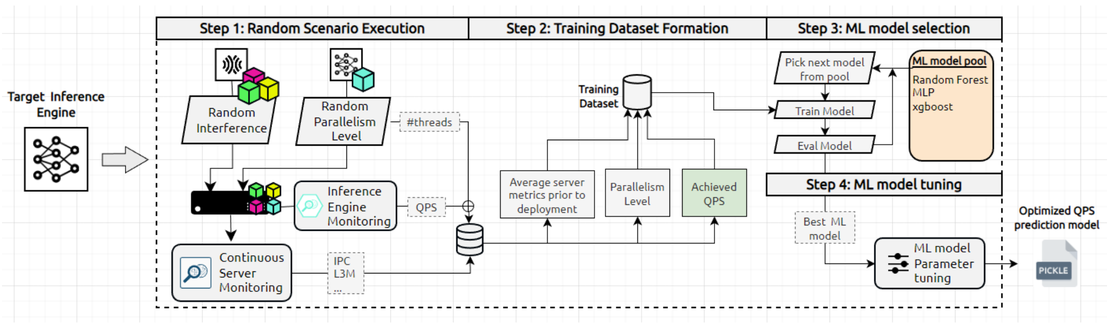

Step 1: Random Scenario Execution
Step 2: Training Dataset Formation
Step 3: ML model selection
Target Inference Engine
Random Interference
Random Parallelism Level
#threads
Inference Engine Monitoring
QPS
Continuous Server Monitoring
IPC
L3M
...
Training Dataset
Average server metrics prior to deployment
Parallelism Level
Achieved QPS
Pick next model from pool
Train Model
Eval Model
ML model pool
Random Forest
MLP
xgboost
Step 4: ML model tuning
Best ML model
ML model Parameter tuning
Optimized QPS prediction model
PICKLE

Figure 5.1: Offline Scheduler Diagram of the (1) training scenarios execution, (2) the training dataset formation, (3) the training and evaluation of the ML Regression models on the

dataset, (4) the ML model selection and optimization, for each MLPerf Inference benchmark (Target Inference Engine)

Each training scenario consists of the deployment of a random number of “iBench” jobs over Kubernetes, each pressuring one, randomly selected, specific shared resource of the cluster (cpu, l2 cache, l3 cache, memory bandwidth or memory capacity) for the duration of the scenario. The total number of deployed iBench jobs in a scenario ranges from 1 to 16. Their total number, as well as the way these iBench jobs are partitioned into pressuring the 5 resources in contention, defines the interference conditions of the scenario. The intensity of the pressure on each shared resource is determined accordingly by the subset of the total number of concurrent iBench jobs that is set to put pressure on that resource. Consequently, the more “iBench” jobs pressuring a resource of the cluster, the stronger interference the cluster encounters on that resource.

After deploying the “iBench” jobs in each training scenario, we collect a set of various system metrics, regarding the cpu and memory usage levels on the cluster, every second, for the duration of 20 seconds, and we take the mean value of each metric over that period of time. As a result, we get a clear image of the current state of our system each time, affected by our deployed sources of interference, before we run our MLPerf Inference benchmark.

The system metrics we collect on our Kubernetes worker node derive from the “*mpstat”* linux command, the “*free”* linux command, as well as from queries to an InfluxDB time series database that resides in a different server, monitoring and keeping various core and memory usage metrics of our worker VM configuration. The “mpstat” command collects and displays performance statistics for all logical processors in the system. The “free” command displays the total amount of free space available along with the amount of memory used and swap memory in the system, and also the buffers used by the kernel.

The following system metrics are collected from the worker node in each training scenario:

Queries	**core metrics**									**memory metrics**	
mpstat (%)	%usr	%sys	%idle								
free (in KB)										used	free
influxDB	c0res	c1res	c3res	c6res	c7res	exec	ipc	l2m	l3m	mem_read	mem_write

Table 5.1: Low-level system metrics collected from the worker node

The final step in our training scenario is the deployment of an MLPerf Inference benchmark, in the current system state. Firstly, we randomly pick one of the MLPerf Inference models, either for the image classification or the object detection task, presented in chapter 3. We exclude from our choice the mobilenet model with the tflite backend, since it does not run on a MultiStream benchmark scenario. If the selected model runs on an onnxruntime backend, we assign random values, ranging from 1 to 8, to the OMP_NUM_THREADS variable and the --threads option in the MLPerf deployment yaml file. In the same manner, if the selected model runs on a tensorflow backend, we assign a random value, ranging from 1 to 8, to the INTRA_OP_PARALLELISM_THREADS option in the MLPerf deployment yaml file. Then, we deploy the MLPerf Inference benchmark over Kubernetes, running the selected model with the selected values for the options mentioned above, in a MultiStream benchmark scenario for a duration of 60 seconds.

When the running of the MLPerf Inference benchmark completes, the QPS value of the benchmark results is pushed to the Prometheus platform for Kubernetes monitoring, so as to be available to be read by all the Kubernetes pods running in the cluster. To accomplish that, we first installed the Prometheus Pushgateway service, a push acceptor, for ephemeral and batch jobs, that will retain the pushed metric so it can be scraped later on.[41] Then, we appended the snippet of code, shown below, to the *run_local.sh* file of the MLPerf Inference Benchmarks suite for image classification and object detection tasks.

```
RESULT=$(cat $OUTPUT_DIR/results.json | jq '. | \
{qps: ."TestScenario.MultiStream".qps} | .qps')
echo "some_metric $RESULT" | curl --data-binary \
@- 10.100.7.187:9091/metrics/job/some_job
```

Having the Prometheus Pushgateway service as a target to scrape the achieved QPS of the benchmark, we create a csv file, where each row represents a single training scenario. Each row contains the system metrics collected at the beginning of each training scenario, the assigned values of the INTRA_OP_PARALLELISM_THREADS (for the tensorflow backend), OMP_NUM_THREADS and --threads options (for the onnxruntime backend), the name of the selected MLPerf Inference model, the backend used and the resulting QPS score of the benchmark.

The csv file we obtain from these training scenarios represents the training dataset of our Machine Learning Regression Models, where the dependent variables are the system metrics, together with the INTRA_OP_PARALLELISM_THREADS, OMP_NUM_THREADS and --threads options, and the independent variable is the QPS. We then separate our training dataset into the independent (y) and dependent (X) variables, we isolate the rows regarding a single MLPerf Inference benchmark every time and train a set of various Regression Models, each with its **default training parameters**. Finally, we compare the accuracy of their prediction of QPS using the score by cross-validation of the scikit-learn python library[42].

We use the 10-fold cross validation splitting strategy (cv=10) and the estimator's default score method is used each time. **All scorer objects follow the convention that higher return values are better than lower return values[43]**. A negative score means that the model fitted the data extremely badly. To evaluate the performance of an ML Regression estimator, the following procedure is followed for each MLPerf Inference benchmark.

The training set is split into 10 smaller sets. Then, for each of the 10 "folds", the ML Regression model is trained using 9 of the folds as training data, and the resulting model is validated on the remaining part of the data (i.e., it is used as a test set to compute a performance measure such as accuracy). The performance measure reported by the 10-fold cross-validation is then the average of the values computed in the loop.

The exploration of the accuracy scores of various ML Regression Models is presented in the following tables:

[41] https://sysdig.com/blog/kubernetes-monitoring-with-prometheus-alertmanager-grafana-pushgateway-part-2/
[42] https://scikit-learn.org/stable/modules/cross_validation.html
[43] https://scikit-learn.org/stable/modules/model_evaluation.html#scoring-parameter

MLPerf Inference Benchmarks (image classification task)					
Algorithm	**onnx-resnet50**	**tf-resnet50**	**onnx-mobilenet**	**tf-mobilenet**	**tf-mobilenet-quant**
Linear	-1.3e+12	-5.4e+23	-4.8e+11	-1.2e+23	0.813
Ridge	-6.1e+11	-1.6e+12	-1.6e+11	-2.9e+12	0.814
Lasso	0.679	0.745	0.801	0.783	0.797
Elastic Net	0.679	0.744	0.803	0.783	0.798
Bayesian Ridge	-3e+9	-4.2e+10	-1.7e+11	-9.2e+11	0.812
SGD	-5.1e+43	-6.4e+43	-1.8e+43	-5.1e+42	-5.1e+42
SVR	0.253	0.201	0.295	0.232	0.202
k-NN	0.236	0.19	0.324	0.239	0.266
Gaussian Process	-3	-1.9	-3.5	-2.1	-1.6
Decision Tree	0.632	0.921	0.835	0.93	0.92
Random Forest	0.825	0.955	0.909	0.962	0.959
MLP	-3e+8	-4e+8	-1.2e+8	-3.6e+7	-5.6e+6
XGBoost	0.843	0.962	0.914	0.962	0.956

Table 5.2: Score by 10-fold cross-validation of different ML Regression Models for the prediction of the QPS of various MLPerf Inference benchmarks performing an image classification task.

MLPerf Inference Benchmarks (object detection task)				
Algorithm	**onnx-ssd-mobilenet**	**tf-ssd-mobilenet**	**tf-ssd-mobilenet-quant-finetuned**	**tf-mobilenet-symmetr-quant-finetuned**
Linear	-1.7e+13	0.731	-1.1e+12	-6.4e+9
Ridge	-8.9e+12	0.767	-6.7e+11	-8.4e+10
Lasso	0.838	0.792	0.763	0.769
Elastic Net	0.842	0.791	0.762	0.765
Bayesian Ridge	-1.3e+13	0.797	-6.6e+11	-6.7e+10
SGD	-1.3e+45	-2e+43	-2.6e+43	-2e+43
SVR	0.332	0.132	0.139	0.772
k-NN	0.327	0.134	0.126	0.55
Gaussian Process	-2.4	-2	-2.2	-2.1
Decision Tree	0.878	0.927	0.926	0.929
Random Forest	0.942	0.963	0.954	0.955
MLP	-1.4e+10	-1.3e+5	-8.4e+8	-2.3e+6
XGBoost	0.941	0.965	0.956	0.961

Table 5.3: Score by 10-fold cross-validation of different ML Regression Models for the prediction of the QPS of various MLPerf Inference benchmarks performing an object detection task.

As we can tell from the above tables, the most accurate ML Regression Models in the prediction of the performance of the MLPerf Inference benchmarks are the Random Forest Regression and the XGBoost regression. This is true for both the image classification task and the object detection task. It is also apparent that the XGBoost Regression, in particular, has the best score overall in almost all of the predictions. Since the Random Forest Regression outperforms the XGBoost Regression in only 2 of the total 9 predictions, and only by little, for the sake of simplicity, we are going to use the XGBoost Regressor for the prediction of QPS for all the MLPerf Inference benchmarks.

In order to further increase the score by cross-validation we obtained from the XGBoost Regressor at the default configuration, we perform a hyperparameter optimization process to choose a set of hyperparameters for our regressor, tailored to the performance of each MLPerf Inference benchmark it has to predict.

We perform a randomized search on the hyperparameters of the XGboost Regressor for each MLPerf Inference benchmark, using the RandomizedSearchCV function of scikit-learn python library[44]. We specify a range of different values for a set of various parameters that control the learning process of the algorithm.

```
#For XGBRegressor:

param_grid = {
        'silent': [False],
        'max_depth': [6, 10, 15, 20],
        'learning_rate': [0.001, 0.01, 0.1, 0.2, 0,3],
        'subsample': [0.5, 0.6, 0.7, 0.8, 0.9, 1.0],
        'colsample_bytree': [0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0],
        'colsample_bylevel': [0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0],
        'min_child_weight': [0.5, 1.0, 3.0, 5.0, 7.0, 10.0],
        'gamma': [0, 0.25, 0.5, 1.0],
        'reg_lambda': [0.1, 1.0, 5.0, 10.0, 50.0, 100.0],
        'n_estimators': [100, 200, 400, 800, 1600, 3200]}
```

A fixed number of parameter settings is then sampled from the specified distributions to be tried out, given by the "n_iter" function argument, creating a tradeoff between runtime and quality of the solution. In our hyperparameter tuning we use n_iter=100, so the model is fit 100 times for different combinations of the parameters from the specified distribution. We also use a 5-fold cross-validation strategy (cv=5), and the scoring strategy, to evaluate the performance of the 5-fold cross-validated model on each test set of the optimization process, is set to "neg_mean_squared_error" for regression.

The resulting improvement on the accuracy score by the 10-fold cross-validation of the XGBoost Regression algorithm for each MLPerf Inference benchmark is presented in the following table.

MLPerf Inference benchmarks	XGBoost Regression (default parameters)	XGBoost Regression (tuned parameters)
onnxruntime-resnet50	0.843	0.877
tensorflow-resnet50	0.962	0.964

[44] https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html

onnxruntime-mobilenet	0.914	0.924
tensorflow-mobilenet	0.962	0.97
tensorflow-mobilenet-quantized	0.956	0.966
onnxruntime-ssd-mobilenet	0.941	0.943
tensorflow-ssd-mobilenet	0.965	0.971
tensorflow-ssd-mobilenet-quantized-finetuned	0.956	0.964
tensorflow-ssd-mobilenet-symmetrically-quantized-finetuned	0.961	0.966

Table 5.4: Score by 10-fold cross-validation comparison of the XGBoost Regression algorithm prior and after hyperparameter optimization

The values of the parameters, which were selected by the RandomizedSearchCV function as the best at each case for the XGBoost Regression algorithm, are the following:

MLPerf Inference benchmarks	**Best parameters for XGBoost Regression**
onnxruntime-resnet50	{'subsample': 0.7, 'silent': False, 'reg_lambda': 100.0, 'n_estimators': 1600, 'min_child_weight': 1.0, 'max_depth': 10, 'learning_rate': 0.2, 'gamma': 0.5, 'colsample_bytree': 0.8, 'colsample_bylevel': 1.0}
tensorflow-resnet50	{'subsample': 0.8, 'silent': False, 'reg_lambda': 50.0, 'n_estimators': 400, 'min_child_weight': 5.0, 'max_depth': 15, 'learning_rate': 0.1, 'gamma': 0.5, 'colsample_bytree': 1.0, 'colsample_bylevel': 0.7}
onnxruntime-mobilenet	{'subsample': 0.7, 'silent': False, 'reg_lambda': 5.0, 'n_estimators': 1600, 'min_child_weight': 0.5, 'max_depth': 6, 'learning_rate': 0.01, 'gamma': 1.0, 'colsample_bytree': 0.6, 'colsample_bylevel': 0.9}
tensorflow-mobilenet	{'subsample': 0.5, 'silent': False, 'reg_lambda': 50.0, 'n_estimators': 3200, 'min_child_weight': 1.0, 'max_depth': 20, 'learning_rate': 0.1, 'gamma': 0.5, 'colsample_bytree': 1.0, 'colsample_bylevel': 0.9}
tensorflow-mobilenet-quantized	{'subsample': 0.8, 'silent': False, 'reg_lambda': 50.0, 'n_estimators': 3200, 'min_child_weight': 5.0, 'max_depth': 15, 'learning_rate': 0.01, 'gamma': 0.5, 'colsample_bytree': 0.9, 'colsample_bylevel': 0.8}
onnxruntime-ssd-mobilenet	{'subsample': 0.5, 'silent': False, 'reg_lambda': 5.0, 'n_estimators': 800, 'min_child_weight': 10.0, 'max_depth': 20, 'learning_rate': 0.1, 'gamma': 0.5, 'colsample_bytree': 0.8, 'colsample_bylevel': 0.8}
tensorflow-ssd-mobilenet	{'subsample': 1.0, 'silent': False, 'reg_lambda': 50.0, 'n_estimators': 3200, 'min_child_weight': 5.0, 'max_depth': 6, 'learning_rate': 0.1, 'gamma': 0.25, 'colsample_bytree': 1.0, 'colsample_bylevel': 0.9}

tensorflow-ssd-mobilenet-quantized-finetuned	{'subsample': 0.8, 'silent': False, 'reg_lambda': 5.0, 'n_estimators': 100, 'min_child_weight': 3.0, 'max_depth': 15, 'learning_rate': 0.1, 'gamma': 0, 'colsample_bytree': 1.0, 'colsample_bylevel': 0.7}
tensorflow-ssd-mobilenet-symmetrically-quantized-finetuned	{'subsample': 0.9, 'silent': False, 'reg_lambda': 10.0, 'n_estimators': 3200, 'min_child_weight': 1.0, 'max_depth': 10, 'learning_rate': 0.01, 'gamma': 0.5, 'colsample_bytree': 1.0, 'colsample_bylevel': 0.5}

Table 5.5: Best parameters for the XGBoost ML Regression algorithm for each MLPerf Inference benchmark, after hyperparameter tuning

To visualize the accuracy of the predictions of the optimized XGBoost Regression model for each MLPerf Inference benchmark, we perform the following experiment. We split the training dataset of each benchmark, by holding out a small part of the available data as a test set X_test, y_test, and we train the corresponding ML model on the remaining training set. Then, we evaluate the ML model on the test set by comparing its predictions y_pred (of the X_test) to the y_test value, as shown in the figure below.

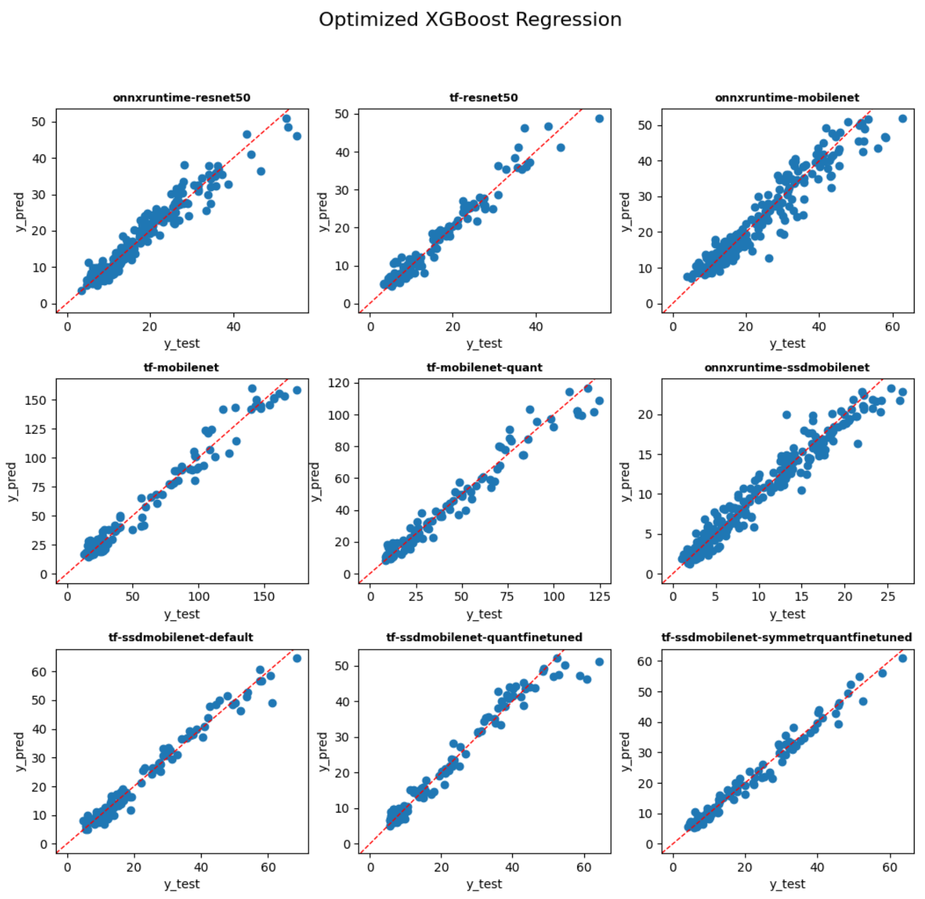


Figure 5.2: Scatter plot: Accuracy of predictions of the optimized XGBoost Regression model for each MLPerf Inference benchmark. The red line has an angle of 45 degrees, representing the desired output, where y_pred=y_test.

As we see from the figure, the predictions of the optimized XGBoost Regression model for the MLPerf Inference benchmarks fall around the 45-degree line, with small deviations from it, meaning the ML model, with its newly-tuned parameters, is well-trained to tackle the tasks ahead.

We save the resulting ML Regression model, to use it for the prediction of the QPS score of all our MLPerf Inference benchmarks in our experiments.

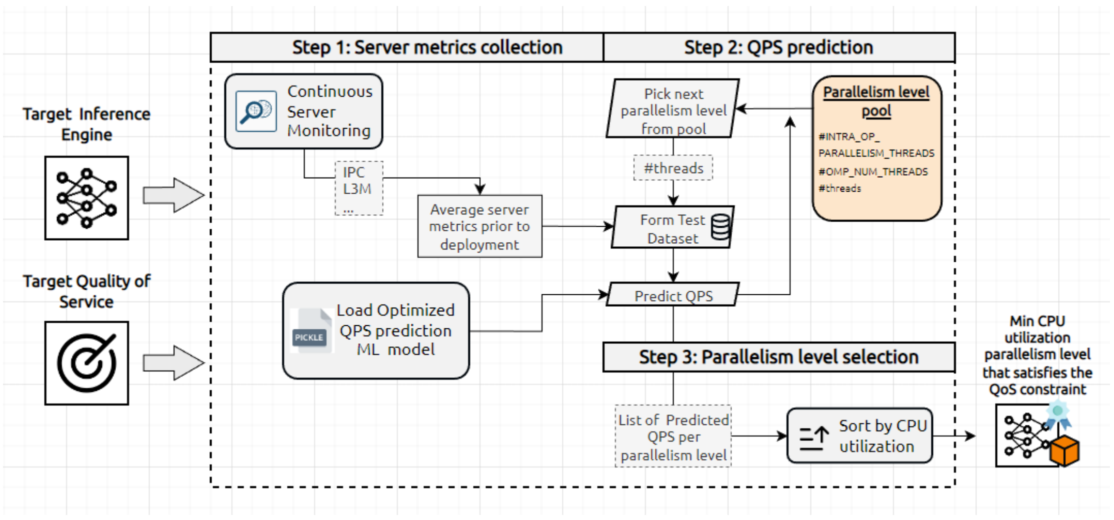
5.2 Online: Interference and resource aware predictive inference engine scheduler

In this section, we analyze the algorithm that our custom, model-specific Inference Engine scheduling mechanism uses, as well as the algorithm of our model-less Inference Engine scheduler design.

5.2.1 Model-Specific Deployment

The custom, model-specific Inference Engine scheduling mechanism we developed is built to take into account the current level of interference conditions in the system before deploying the MLPerf Inference benchmark on a MultiStream scenario to run in the cluster. It also aims to serve the benchmark with a configuration (parallelism level) that allows it to be as little resource intensive to the system as possible while still managing to satisfy a required QoS constraint.

The exact mechanism of our custom, model-specific scheduler is presented below. The steps include: (a) The collection of system metrics for the current state of the cluster, (b) the QPS prediction of the MLPerf inference benchmark (target inference engine) in that cluster state, by the loaded optimized ML model, for each parallelism level, (c) the selection of the least CPU intensive parallelism level that satisfies the QoS constraint, (d) and the final deployment of the inference engine on the cluster, with the selected parallelism level.



Step 1: Server metrics collection
Step 2: QPS prediction
Target Inference Engine
Target Quality of Service
Continuous Server Monitoring
IPC
L3M
...
Average server metrics prior to deployment
Load Optimized QPS prediction ML model
PICKLE
Pick next parallelism level from pool
#threads
Form Test Dataset
Predict QPS
Parallelism level pool
#INTRA_OP_PARALLELISM_THREADS
#OMP_NUM_THREADS
#threads
Step 3: Parallelism level selection
List of Predicted QPS per parallelism level
Sort by CPU utilization
Min CPU utilization parallelism level that satisfies the QoS constraint

Figure 5.3: Online Custom, Model-Specific Scheduler Diagram of the parallelism level selection process of a target inference engine, in order to satisfy a target QoS with minimum resources utilization.

The custom, model-specific scheduler takes as input the name of the MLPerf Inference benchmark that is going to perform either an image classification or an object detection task. It also receives a desired QPS score that it aims to have as a lower bound for the performance of the scheduled benchmark. This lower-bounded performance request would serve as a QoS constraint in our scheduling mechanism.

When a scheduling request comes to our custom scheduler, it immediately collects a set of various system metrics, regarding the current CPU and memory usage levels on the cluster, every second, for a set duration of time, and it keeps the mean value of each metric over that period of time. These metrics, about the current state of the Kubernetes cluster, derive from the same queries that were used in the training of the ML Regression model, which our scheduler is going to utilize.

After receiving the current system metrics, our scheduler loads the saved XGBoost Regression model that has been trained for the MLPerf Inference benchmark that is to be scheduled. Then, it executes the following procedure:

Depending on the backend (tensorflow or onnxruntime) the benchmark will use, it sets a value to the corresponding variables (INTRA_OP_PARALLELISM_THREADS for the tensorflow backend, OMP_NUM_THREADS and --threads option for the onnxruntime backend) of the benchmark. The scheduler asks the Regression model to predict the QPS score the benchmark would achieve, if it run with that parallelism level, under the interference conditions indicated by the collected system metrics.

It repeats that procedure for all the possible value combinations of the suitable variables of the benchmark (each value ranging from 1 to 8), storing all the QPS predictions of the Regression model (one QPS prediction for each unique configuration of the benchmark, under the same current system state), as well as the values of the benchmark variables that led to that prediction, in an array.

After filling the array, it sorts it in two different ways. Firstly, it sorts it by the value of the predicted QPS score, and saves the parallelism level of the benchmark that presumably achieves the highest score in the current system state. The scheduler uses that configuration if the QoS constraint is higher than the highest predicted QPS score, so that the benchmark will run at the best performance possible.

If the QoS constraint is lower than, or equal to the highest predicted QPS score, the scheduler proceeds in sorting the array by the amount of CPU utilization each entry has, in an ascending order. Thereupon, it selects the parallelism level of the benchmark that will potentially result in a QPS score that satisfies the QoS constraint (based on the value of the predicted QPS for each parallelism level in the array) with the lowest possible CPU utilization.

The duration that a scheduled MLPerf Inference benchmark will run on the cluster for, as well as the duration for which the custom scheduler will collect the metrics of the system, prior to scheduling the benchmark, are both configurable.

5.2.2 Model-less approach

In the model-less approach to our custom Inference Engine scheduling mechanism, we introduce a new model-less interface, where developers need to specify only the task that they want to execute (image classification, object detection) and the high-level performance they require as a target QoS. The model-less scheduler, then, selects the best Inference Engine, from a task-specific pool of registered, trained Inference models, with the appropriate parallelism level to accomplish that job in the least resource intensive way, while aiming to satisfy the QoS constraint under the current level of interference conditions in the system.

The steps our custom, model-less Inference Engine scheduler takes to deploy an inference model in the Kubernetes cluster are the following: (a) The collection of system metrics for the current state of the cluster, (b) the QPS prediction, for each parallelism level, of an inference model, in that cluster state, by the corresponding loaded optimized ML model, (c) the selection of the least CPU intensive parallelism level for that inference model, that satisfies the QoS constraint, (d) the repetition of steps (b) and (c) for all the inference engines registered for the requested task, (e) the selection of the {inference engine - parallelism level} pair that satisfies the QoS constraint with the least CPU load, (f) and the final deployment of the selected inference engine on the cluster, with the selected parallelism level.

The custom, model-less scheduler takes as input the name of the task, that the MLPerf Inference benchmarks are going to perform, which is either image classification or object detection. It also receives a target QoS, as a lower bound for the performance of the scheduled benchmark.

When a scheduling request comes to our model-less scheduler, similarly to the first step of the model-specific scheduler, it immediately collects a set of various system metrics, regarding the current CPU and memory usage levels on the cluster, every second, for a set duration of time, and it keeps the mean value of each metric over that period of time.

After receiving the current system metrics, the model-less scheduler navigates the space of the registered inference models for the requested task. Then, for each inference model there, it proceeds in a similar manner to the model-specific scheduler, until it has the parallelism level, and the corresponding QPS prediction, of each inference model, that would presumably satisfy the QoS constraint in the least CPU intensive way, under the current interference conditions.

Subsequently, the model-less scheduler stores all the triples {inference engine – selected parallelism level – predicted QPS in that parallelism level} in a list, one entry for each inference engine that is registered for the requested task. It sorts the list by the amount of CPU utilization each entry has, in ascending order. Finally, it selects to deploy the inference engine, at the parallelism level it was paired with in the triple, that will potentially achieve a QPS score that satisfies the QoS constraint (based on the value of the predicted QPS score, paired with the inference engine in the triple) with the lowest possible CPU utilization.

In a similar way to the model-specific scheduler, the duration that a scheduled Inference Engine will run on the cluster for, as well as the duration for which the model-less scheduler will collect the metrics of the system, prior to scheduling the inference engine, are both configurable.

Chapter 6

Evaluation

In this chapter, we use our experimental infrastructure to evaluate our custom model-specific Inference Engine scheduling mechanism in a set of different interference scenarios. In addition, we compare our proposed model-specific approach with the results of a scheduler serving the MLPerf Inference benchmarks to run either with minimum or maximum resources utilization, in the same interference scenarios. Finally, we evaluate and compare our model-specific Inference Engine scheduler to our model-less approach of the inference serving, in one interference scenario.

6.1 Scenarios Description

In order to evaluate our custom model-specific Inference Engine scheduler, as well as our model-less Inference Engine scheduler, we generated 3 different interference scenarios to put stress of varying intensity to the shared resources of the system while we executed our experiments.

Each interference scenario consists of the deployment of a random number of "iBench" jobs over Kubernetes, each pressuring one, randomly selected, specific shared resource of the cluster (cpu, l2 cache, l3 cache, memory bandwidth or memory capacity). In the training scenario used for the training of the ML Regression models, the total iBench jobs were deployed at the beginning of the scenario, and they were present for the whole duration of the scenario. In contrast to that, in the interference scenarios, there is a new set of 0 to 2 iBench jobs being deployed after a random amount of time (10s to 30s) has passed from the previously deployed set of iBench jobs. The iBench jobs in each set, put pressure on a randomly selected shared resource, and the set lasts for a random amount of time (70s to 220s) before it is deleted.

This modification in the way the iBench jobs are being deployed, results in an interference scenario that more closely mimics a real-world situation, where multiple different sources of interference can be present, each lasting for a different amount of time, while new workloads (SoIs) begin pressuring the system, together with the current ones, and other SoIs complete their work and stop.

With the intention of having reproducible interference scenarios, so as to compare the behavior of different Inference Engine schedulers in each one of them, we logged the order each set of iBench jobs came, the shared resources the iBench jobs were putting stress on, as well as all the time variables for each set of iBench jobs in a file.

Subsequently, we created 3 different interference scenarios for the cluster, which can easily be replicated, and we used them to run our experiments upon.

6.2 Experiments Description

6.2.1 Model-Specific Inference Engine Scheduler

For our first set of experiments, we use our custom, model-specific Inference Engine scheduler, a scheduler that serves the MLPerf Inference benchmarks to run at maximum CPU utilization, and a scheduler that serves them to run at minimum CPU utilization.

We measured the average amount of time required for the algorithm of the model-specific Inference Engine scheduler to select the parallelism level, that meets the QoS constraint with the least CPU utilization, of the next inference engine deployment.

Framework	**Average Selection Time (s)**
Tensorflow	1.08
Onnxruntime	1.24

Table 6.1: Average time required for the selection of the next configuration for deployment, by the model-specific inference engine scheduler

The difference in the selection time of the algorithm between the two backend frameworks is due to the larger QPS prediction array, the onnxruntime benchmarks have, as we control 2 variables for their parallelism level, instead of 1 for the tensorflow benchmarks.

The percentage of total CPU utilization (%usr + %sys)[45] that each combination of the INTRA_OP_PARALLELISM_THREADS (tensorflow backend) or the OMP_NUM_THREADS and --threads (onnxruntime backend) of the benchmark has, was tested in the cluster with no external SoIs, where a benchmark ran for a set amount of time, during which we collected the core metrics of the system using the "mpstat" Linux command. The resulting mapping of each parallelism level to a CPU utilization percentage was logged for use by our custom schedulers.

Therefore, the following combinations of values for the benchmarks' variables are used by the min and max CPU usage schedulers.

Scheduler	**framework**	**INTRA_OP_PARALLELISM_THREADS**	**OMP_NUM_THREADS**	**--threads**
min_cpu_usage	tensorflow	1		
	onnxruntime		1	1
max_cpu_usage	tensorflow	8		
	onnxruntime		8	1

Table 6.2: Combination of values for the MLPerf Inference Benchmarks' variables, used by the min and max CPU usage schedulers

An interesting observation from the mapping procedure we did, is that, when running in a MultiStream scenario, the following combination of values for the benchmarks' variables

[45] %usr: the percentage of CPU utilization that occurred while executing at the user level (application). %sys: the percentage of CPU utilization that occurred while executing at the system level (kernel). https://man7.org/linux/man-pages/man1/mpstat.1.html

matches their default deployment, meaning their deployment without specifying a parallelism level to them.

framework	INTRA_OP_PARALLELISM_THREADS	OMP_NUM_THREADS	--threads
tensorflow	8		
onnxruntime		1	1

Table 6.3: Combination of values for the MLPerf Inference Benchmarks' variables, that match their default parallelism level

These results suggest that the max CPU usage scheduler will run the MLPerf Inference benchmarks with the tensorflow backend at their default parallelism level, whereas the min CPU usage scheduler will run the benchmarks with the onnxruntime backend at their default parallelism level.

In each experiment, we start an interference scenario and, in parallel, we deploy the MLPerf Inference Benchmarks, presented in chapter 3, as Kubernetes pods, by using our custom scheduler. We also request the scheduler that the performance of each benchmark should satisfy a specific QoS constraint. The benchmarks run in a MultiStream scenario, so we exclude the tflite-mobilenet benchmark from our experiments, since it does not work in that scenario. Each MLPerf Inference benchmark is deployed 10 consecutive times by the same scheduler and it runs for a duration of 30 seconds in the cluster each time. Throughout the experiment, the amount of interference in the cluster changes in accordance with the chosen interference scenario. Our scheduler collects the metrics of the system for 5 seconds, prior to deploying the next benchmark each time.

We repeat each experiment for 3 different QoS constraints, and with the use of 3 different schedulers: our custom scheduler, the min CPU usage scheduler and the max CPU usage scheduler. We also test the schedulers on 3 different interference scenarios and we compare their results in section 6.3.

6.2.2 Model-less Inference Engine Scheduler

For the second set of experiments, we test our model-specific Inference Engine scheduler and our model-less Inference Engine scheduler, using the inference engines from the MLPerf Inference Benchmarks suite, presented in chapter 3, in a single interference scenario. Once more, we exclude the tflite-mobilenet benchmark from our pool, as it does not work in a MultiStream scenario that the benchmarks will run in.

We measured the average amount of time required for the algorithm of the model-less Inference Engine scheduler to select a parallelism level, one for all the potential inference engines for the next deployment, that meets the QoS constraint with the least CPU utilization, and then select the least CPU intensive {inference engine – parallelism level} pair of them.

ML task	**Average Selection Time (s)**
Image Classification	1.86
Object Detection	1.58

Table 6.4: Average time required for the selection of the next inference engine, and its configuration, for deployment, by the model-less inference engine scheduler

The difference in the selection time of the algorithm between the two ML tasks is due to the 5 inference engines for potential deployment that perform the image classification task, as opposed to the 4 inference engines that perform the object detection task. Another contributor reason is that there are 2 onnxruntime benchmarks in the pool of the image classification inference engines, instead of the 1 onnxruntime benchmark in the pool of the object detection inference engines. We recall that the onnxruntime benchmarks have a larger QPS prediction array, as we control 2 variables for their parallelism level, instead of 1 for the tensorflow benchmarks.

In each experiment, conducted to evaluate the model-less scheduler, we start the interference scenario and, in parallel, we specify a task for the MLPerf Inference benchmarks to perform (image classification or object detection). We also request the scheduler that the performance of the deployed benchmark should satisfy a specific QoS constraint. The scheduler makes a selection of what MLPerf Inference benchmark, from the task-specific pool, should run, and it deploys it on the cluster as a Kubernetes pod. We ask the model-less scheduler to select a benchmark 20 consecutive times, and each time the selected benchmark runs in the cluster for a duration of 30 seconds. Throughout the experiment, the amount of interference in the cluster changes in accordance with the chosen interference scenario. Our scheduler collects the metrics of the system for 5 seconds, prior to navigating the space of the registered inference models for the requested task, selecting and deploying the next benchmark each time.

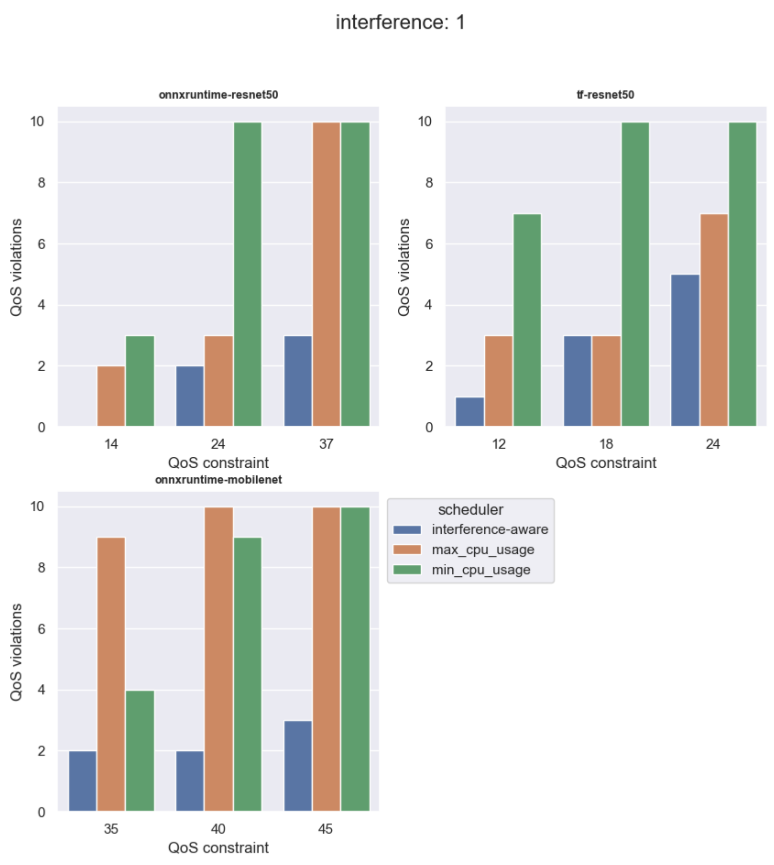
As far as the experiments to evaluate the model-specific scheduler are concerned, they are similar to the first set of experiments, with the following differences: there is a single interference scenario, and our scheduler deploys each MLPerf Inference benchmark 20 consecutive times in the cluster, to run for a duration of 30 seconds each time.

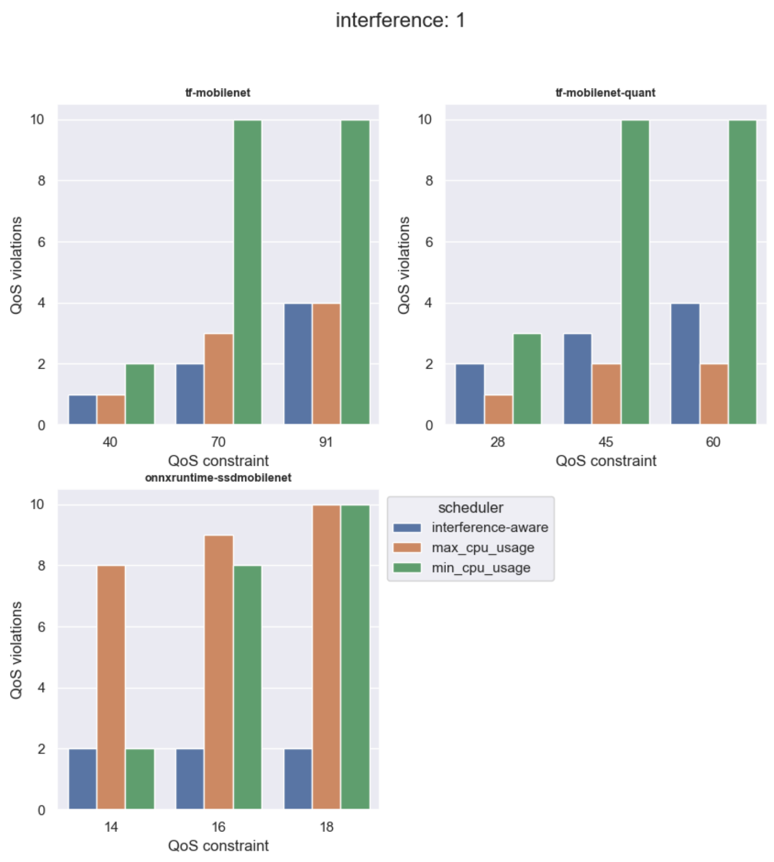
We repeat each experiment for 3 different QoS constraints. The experiments on the model-less design are conducted for 2 different tasks (image classification, object detection). We compare the results of the two schedulers, in section 6.4.

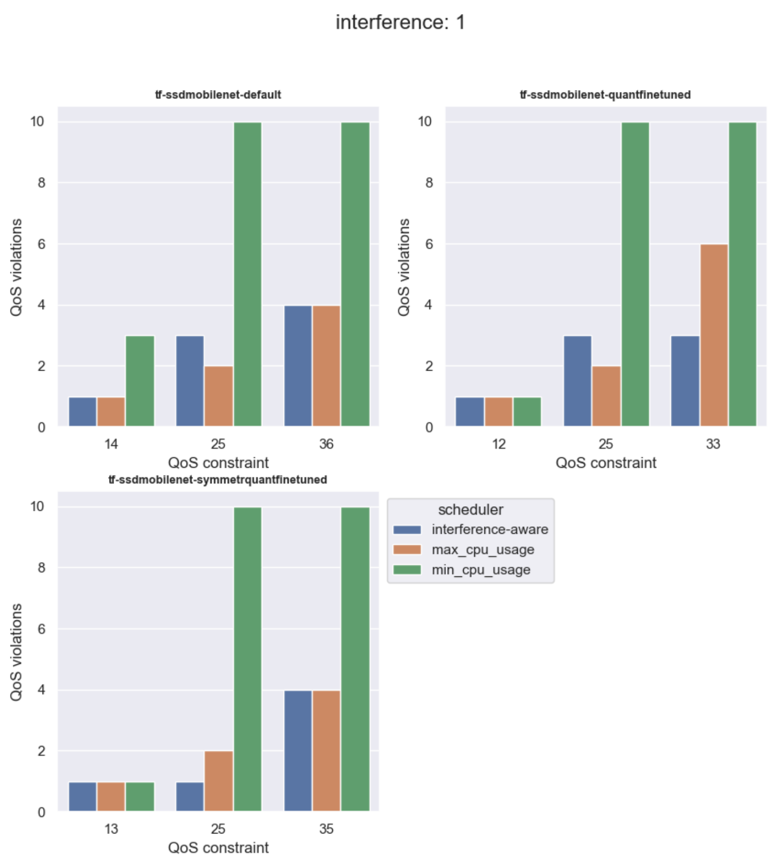
6.3 Model-Specific Inference Engine Scheduler Results & Scheduler Comparison

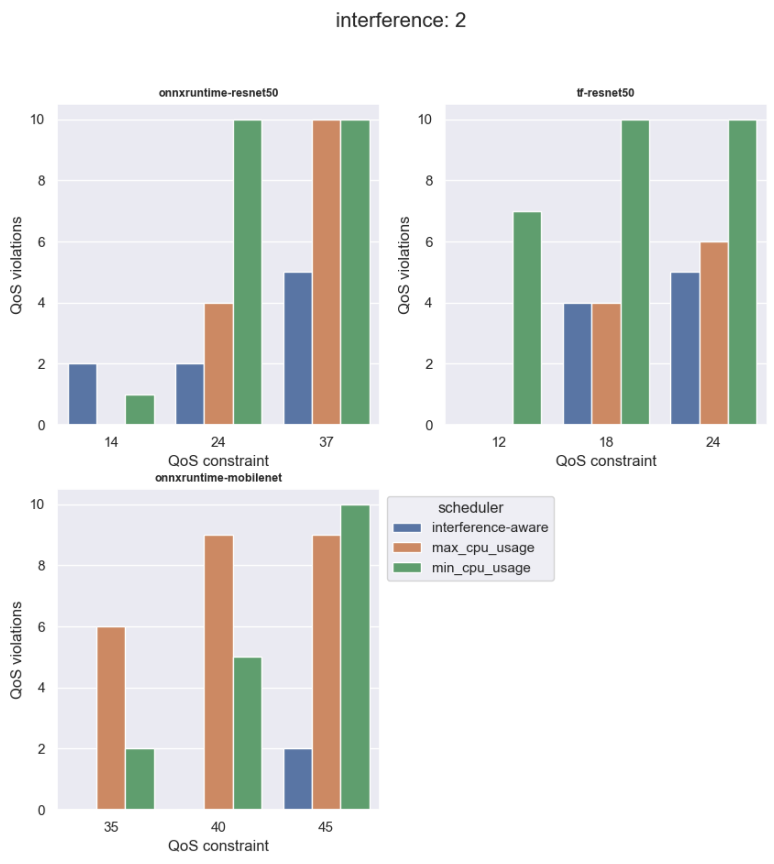
6.3.1 Violations of the QoS constraint

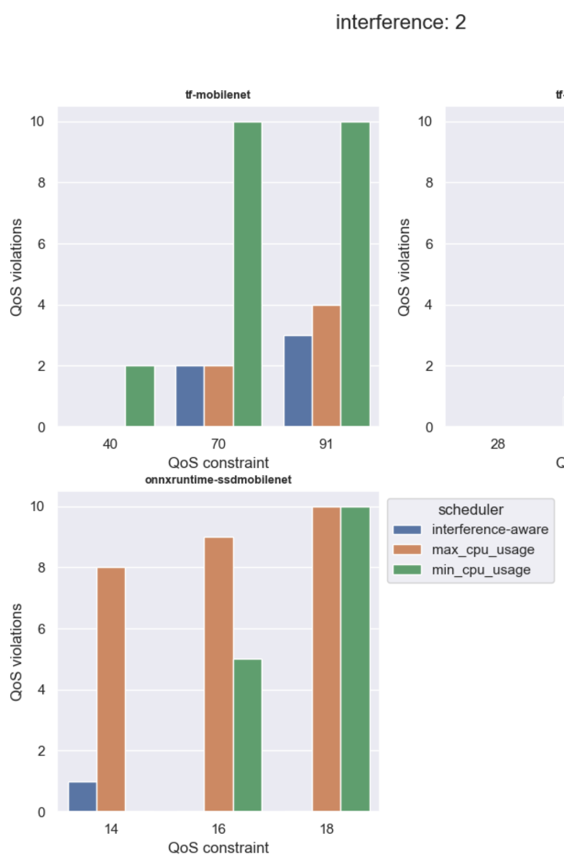
In the first set of figures, we examine the number of times a scheduler did not meet the desired target QoS, but rather it scheduled an MLPerf Inference benchmark with a parallelism level that scored a lower QPS than the QoS constraint. We compare the number of QoS violations between our custom interference-aware scheduler, the min CPU usage scheduler and the max CPU usage scheduler, with 3 different QoS constraints for each MLPerf Inference benchmark, and at 3 different interference scenarios.

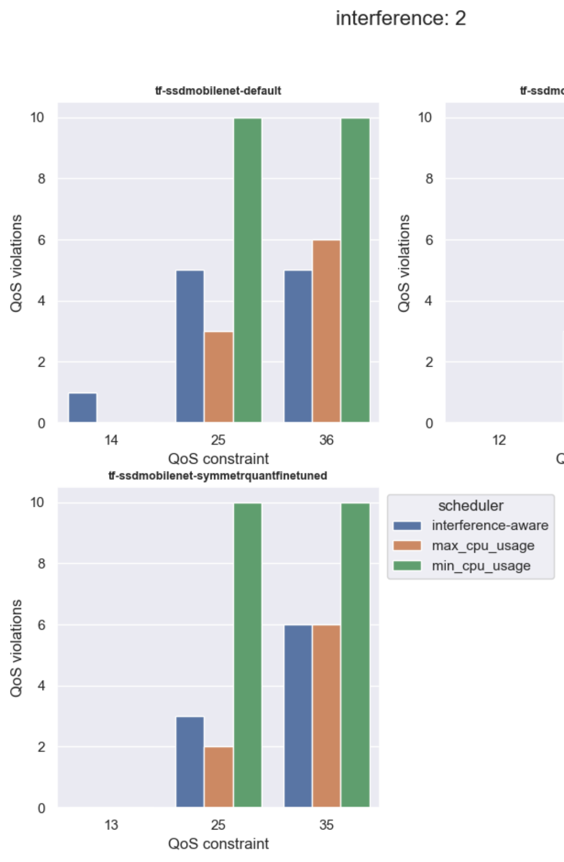

interference: 1
onnxruntime-resnet50
tf-resnet50
onnxruntime-mobilenet
QoS violations
QoS constraint
scheduler
interference-aware
max_cpu_usage
min_cpu_usage

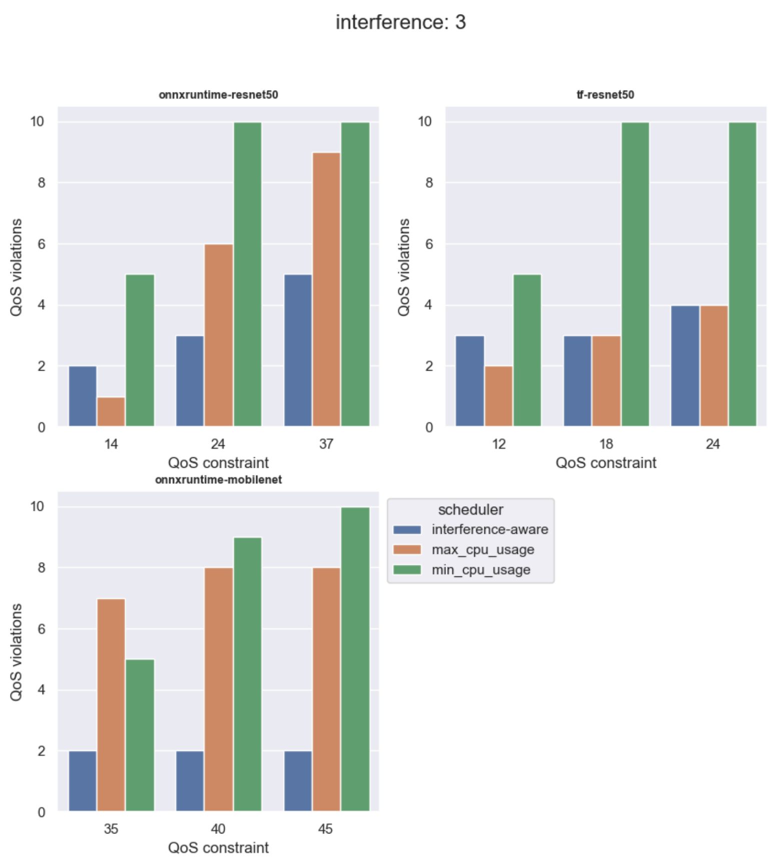

interference: 1
tf-mobilenet
tf-mobilenet-quant
onnxruntime-ssdmobilenet
QoS violations
QoS constraint
scheduler
interference-aware
max_cpu_usage
min_cpu_usage

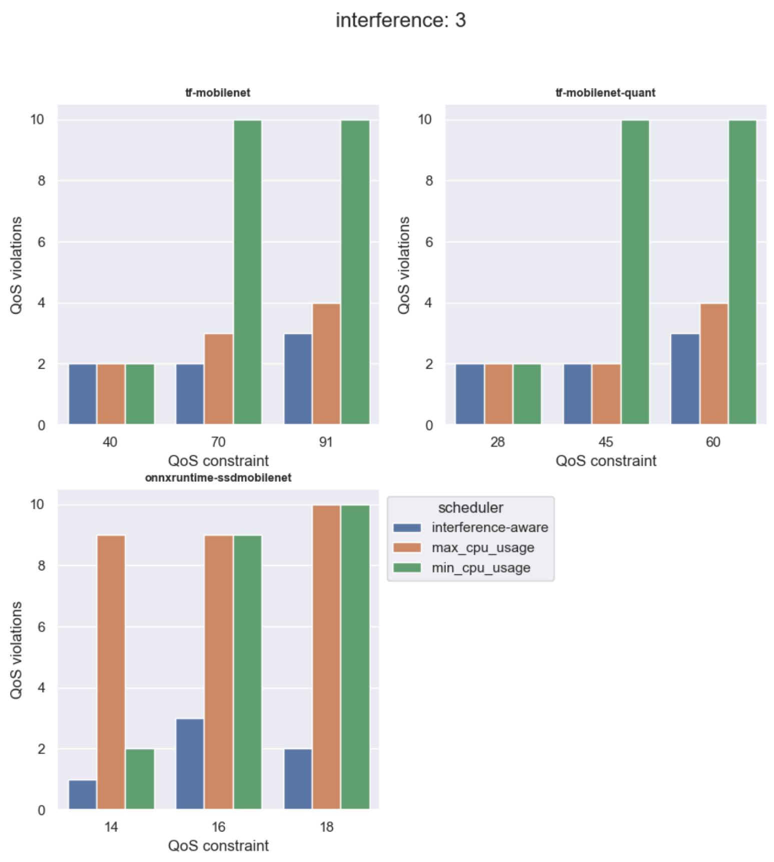

interference: 1
tf-ssdmobilenet-default
tf-ssdmobilenet-quantfinetuned
tf-ssdmobilenet-symmetrquantfinetuned
QoS violations
QoS constraint
14
25
36
12
25
33
13
25
35
scheduler
interference-aware
max_cpu_usage
min_cpu_usage


interference: 2
onnxruntime-resnet50
tf-resnet50
onnxruntime-mobilenet
QoS violations
QoS constraint
14
24
37
12
18
24
35
40
45
scheduler
interference-aware
max_cpu_usage
min_cpu_usage


interference: 2
tf-mobilenet
tf-mobilenet-quant
onnxruntime-ssdmobilenet
QoS violations
QoS constraint
scheduler
🟦 interference-aware
🟧 max_cpu_usage
🟩 min_cpu_usage


interference: 2
tf-ssdmobilenet-default
tf-ssdmobilenet-quantfinetuned
tf-ssdmobilenet-symmetrquantfinetuned
QoS violations
QoS constraint
scheduler
🟦 interference-aware
🟧 max_cpu_usage
🟩 min_cpu_usage


interference: 3
onnxruntime-resnet50
tf-resnet50
onnxruntime-mobilenet
QoS violations
QoS constraint
scheduler
interference-aware
max_cpu_usage
min_cpu_usage


interference: 3
tf-mobilenet
tf-mobilenet-quant
onnxruntime-ssdmobilenet
QoS violations
QoS constraint
scheduler
interference-aware
max_cpu_usage
min_cpu_usage

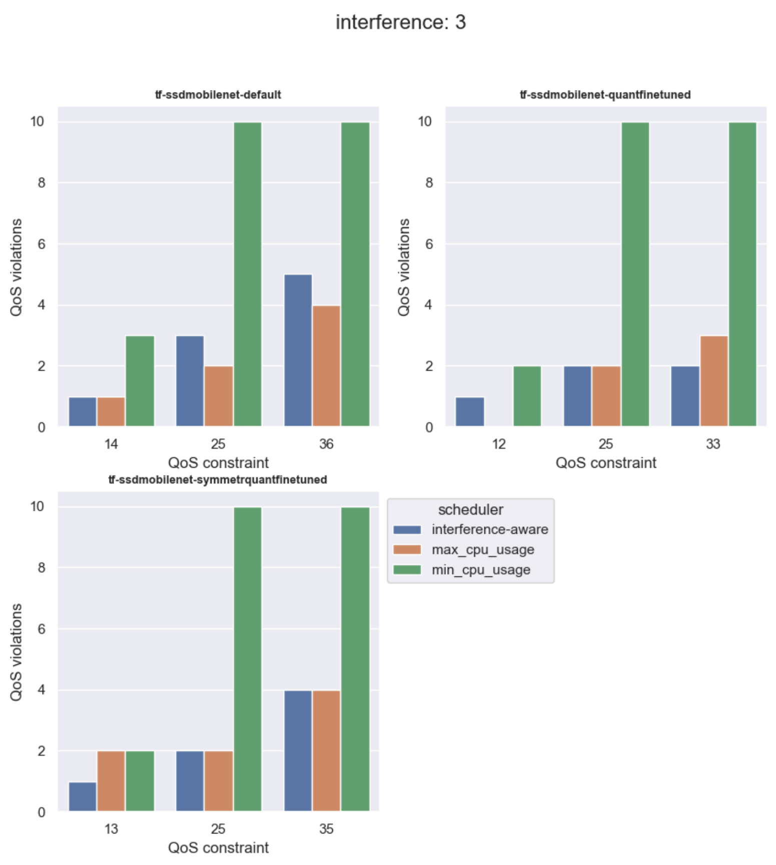


Figure 6.1: Number of QoS violations comparison between the interference-aware scheduler, the max CPU usage scheduler and the min CPU usage scheduler. There are 3 different QoS constraints for each MLPerf Inference benchmark, and 3 different interference scenarios.

It is clear from the figures, that, typically, the lower the QoS constraint is, the better the results are for each scheduler, meaning the number of QoS violations declines. However, we observe that the min CPU usage scheduler cannot keep up with the middle and high QoS constraints, essentially scoring below the QoS constraint for almost all 10 consecutive MLPerf Inference benchmark deployments. This happens because the CPU usage restriction, the benchmarks deployed by that scheduler have, does not allow them to perform very well and meet the higher target QoS.

Our custom interference-aware scheduler appears to perform exceptionally at all 3 QoS constraints on each MLPerf Inference benchmark, in all 3 interference scenarios. It manages to keep the number of QoS violations significantly lower than the min CPU usage scheduler does at all cases on a middle or high QoS constraint, and it performs better than or equal to the min CPU usage scheduler at almost all cases on a low QoS constraint, where it sporadically appears to scarcely surpass the number of QoS violations of the min CPU usage scheduler by 1. Furthermore, our custom scheduler succeeds in performing remarkably better than the max CPU usage scheduler does, on a benchmark with an onnxruntime backend. On the tensorflow benchmarks, our custom scheduler's number of QoS violations is quite similar to the max CPU usage scheduler's one, although our scheduler achieves that without utilizing the resources of the cluster at max capacity all the time.

As far as the min and max CPU usage schedulers are concerned, we notice some interesting behaviors. First of all, for the benchmarks with the tensorflow backend, the max CPU usage scheduler performs great, although at the cost of utilizing a large amount of the resources of the cluster, even when that is not necessary to achieve the target QoS. Nonetheless, for the benchmarks with the onnxruntime backend, the max CPU usage scheduler appears to perform very poorly. When serving the onnxruntime-resnet50 benchmark, the QoS violations are low for a low QoS constraint. However, the max CPU usage scheduler shows more violations as the QoS constraint increases, and it performs the worst at the highest QoS constraint, matching the deficient performance of the min CPU usage scheduler there. An even more unsatisfactory behavior is seen for the onnxruntime-mobilenet and onnxruntime-ssdmobilenet benchmarks, where, not only does the max CPU scheduler perform worse than the min CPU usage scheduler at a low QoS constraint, its number of QoS violations at the low QoS constraint is almost as high as it is at the higher QoS constraints, where the max CPU usage scheduler matches the poor performance of the min CPU usage scheduler.

Despite the substantial number of QoS violations by the max CPU usage scheduler in the onnxruntime benchmarks, our custom interference-aware scheduler seems to perform outstandingly well when serving benchmarks with an onnxruntime backend. By looking at the parallelism levels of the MLPerf Inference benchmarks that our custom scheduler chose at the high QoS constraint, as opposed to the ones the max CPU usage scheduler uses, and checking the mapping of these parallelism levels to the CPU utilization that we had logged, as described in section 6.2, we came to the following conclusions:

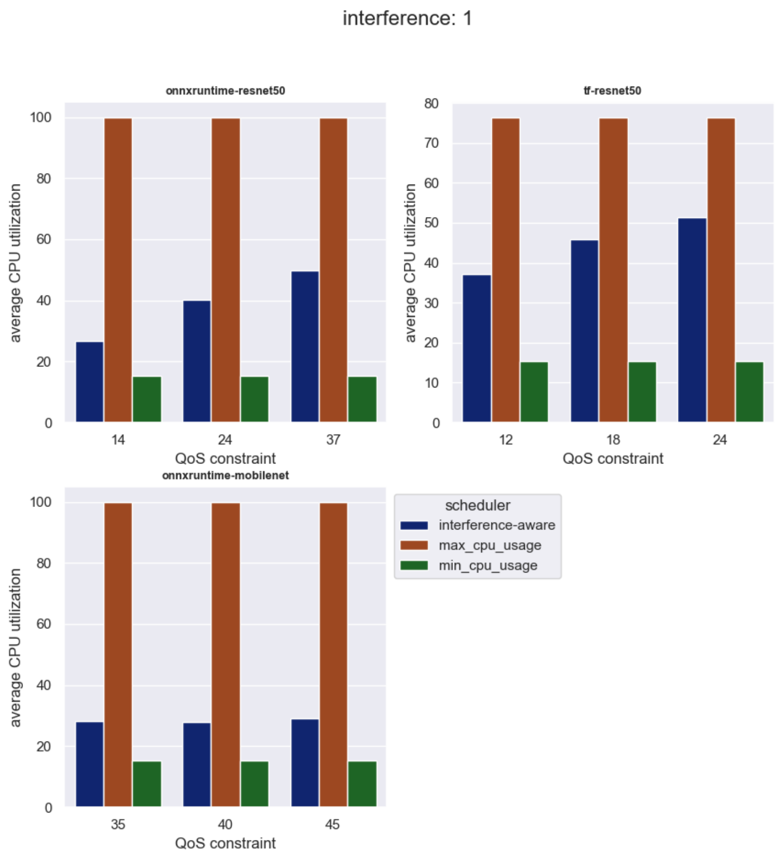
When increasing the parallelism level of the benchmarks with the onnxruntime backend, both the CPU utilization at the user level (application) and the CPU utilization at the system level (kernel) increase in varying amounts, different for each inference engine, depending on the combination of the values of the benchmarks' variables. In general, the higher the value of the CPU utilization at the user level is, the better the performance of the benchmarks.

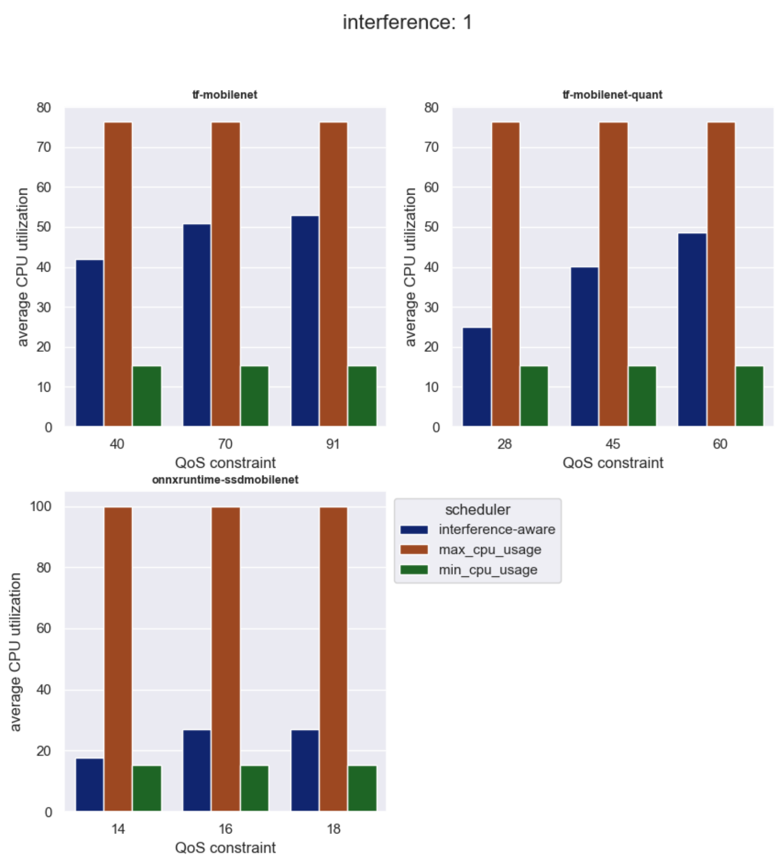
The onnxruntime-resnet50 benchmark performs the best at a total CPU utilization of around 50%, whereas the onnxruntime-mobilenet and onnxruntime-ssdmobilenet benchmarks perform the best at a total CPU utilization of around 30%, under interference. This is because the external interference adds pressure to the CPU, essentially decreasing the available user level time on the CPU for the benchmarks. When that fills up by either the benchmarks or the SoIs, any further increase in the total CPU utilization of the benchmark, imposed by the scheduler, translates to an increase in the CPU utilization at the system level in the CPU, and a decrease at the user level, which creates unnecessary stress to the CPU, and lowers the performance of the inference engines, instead of improving it. This explains the resulting deficient performance of the max CPU usage scheduler on the onnxruntime benchmarks, under interference conditions.

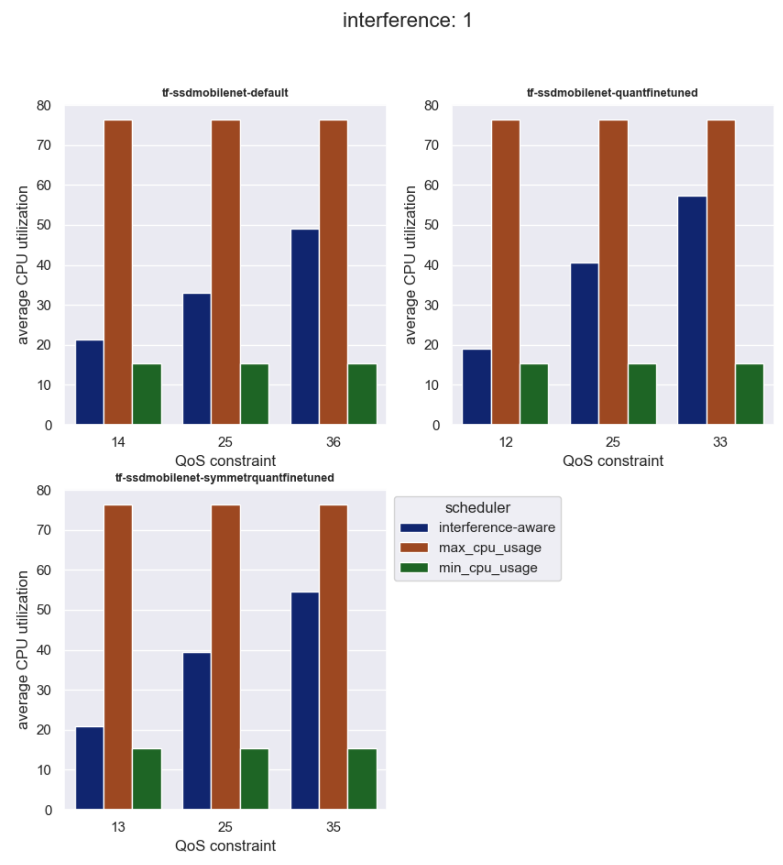
It is worth noting that, when increasing the parallelism level for the benchmarks with the tensorflow backend, the CPU utilization at the system level does not change. Consequently, as the total CPU utilization percentage increases, the QPS scores of the benchmarks get better. When the percentage of CPU utilization at the user level maxes out, either by the benchmarks or the SoIs, there is neither further increase nor decrease in the performance of the benchmarks for higher parallelism levels. This explains their resulting superior performance at a high total CPU utilization, even under interference.

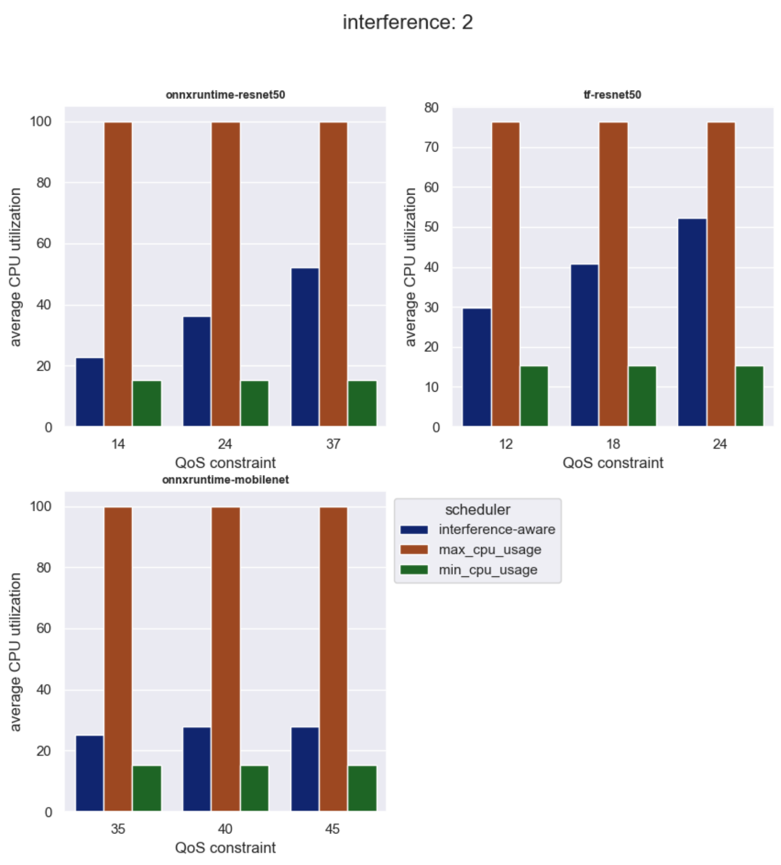
6.3.2 Resources Utilization

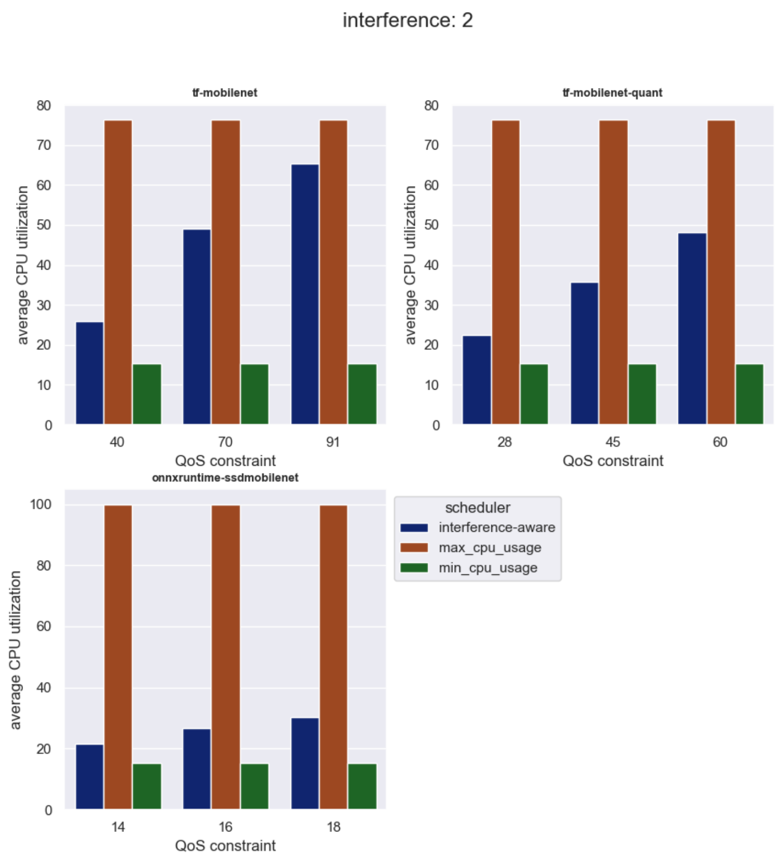
In the next set of figures, we examine what percentage of the resources of the cluster, the deployed MLPerf Inference benchmarks were using, according to their chosen parallelism level each time by the schedulers. More specifically, we compare the average CPU utilization percentage of the 10 consecutive deployments of each MLPerf Inference benchmark on the cluster, between our custom interference-aware scheduler, the min CPU usage scheduler and the max CPU usage scheduler, with 3 different QoS constraints for each benchmark, and at 3 different interference scenarios.



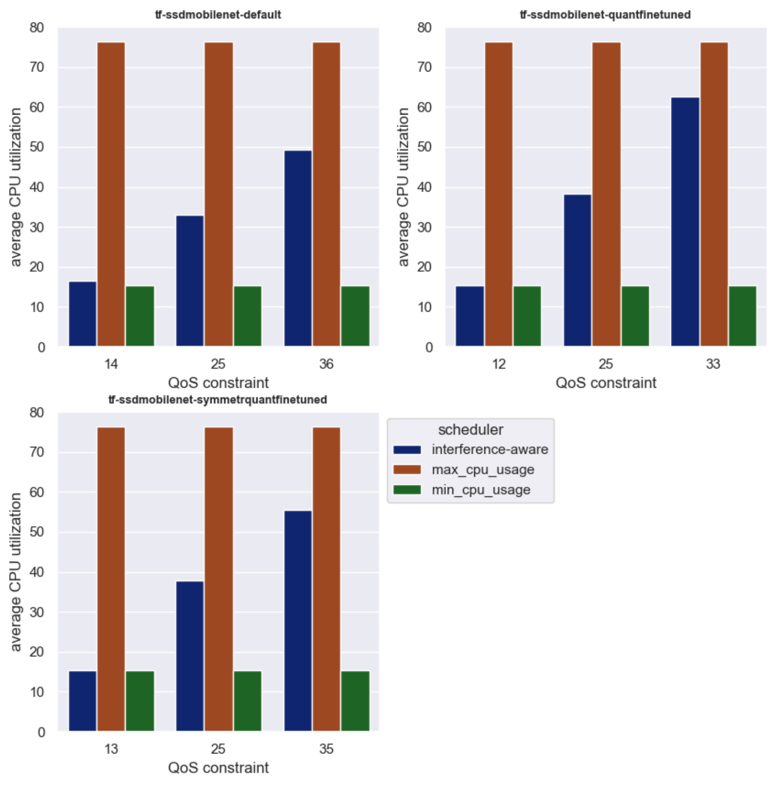

interference: 1
tf-mobilenet
tf-mobilenet-quant
onnxruntime-ssdmobilenet
average CPU utilization
QoS constraint
40
70
91
28
45
60
14
16
18
scheduler
interference-aware
max_cpu_usage
min_cpu_usage


interference: 1
tf-ssdmobilenet-default
tf-ssdmobilenet-quantfinetuned
tf-ssdmobilenet-symmetrquantfinetuned
average CPU utilization
QoS constraint
14
25
36
12
25
33
13
25
35
scheduler
interference-aware
max_cpu_usage
min_cpu_usage

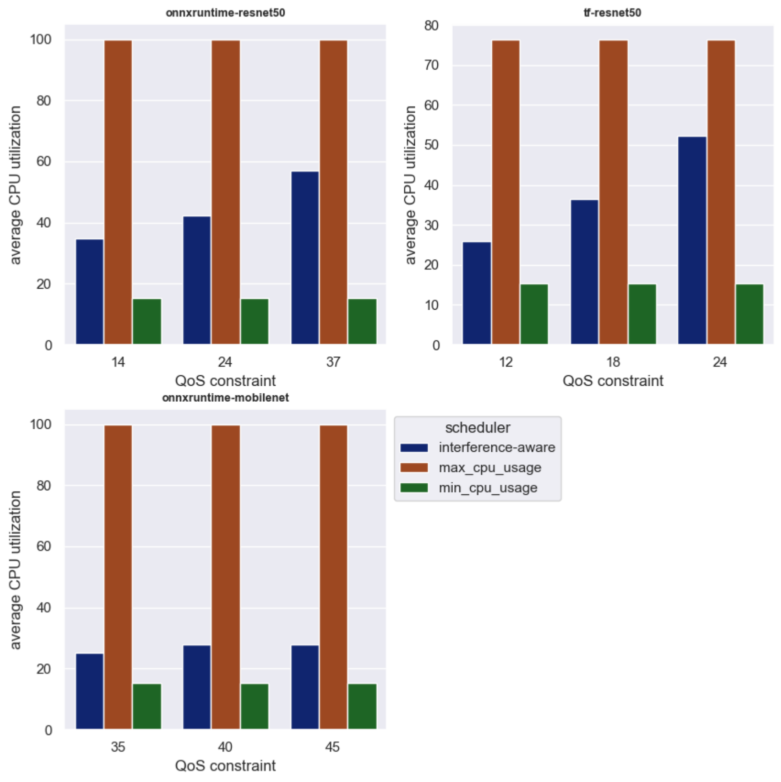

interference: 2
onnxruntime-resnet50
tf-resnet50
onnxruntime-mobilenet
average CPU utilization
QoS constraint
scheduler
interference-aware
max_cpu_usage
min_cpu_usage


interference: 2
tf-mobilenet
tf-mobilenet-quant
onnxruntime-ssdmobilenet
average CPU utilization
QoS constraint
scheduler
interference-aware
max_cpu_usage
min_cpu_usage

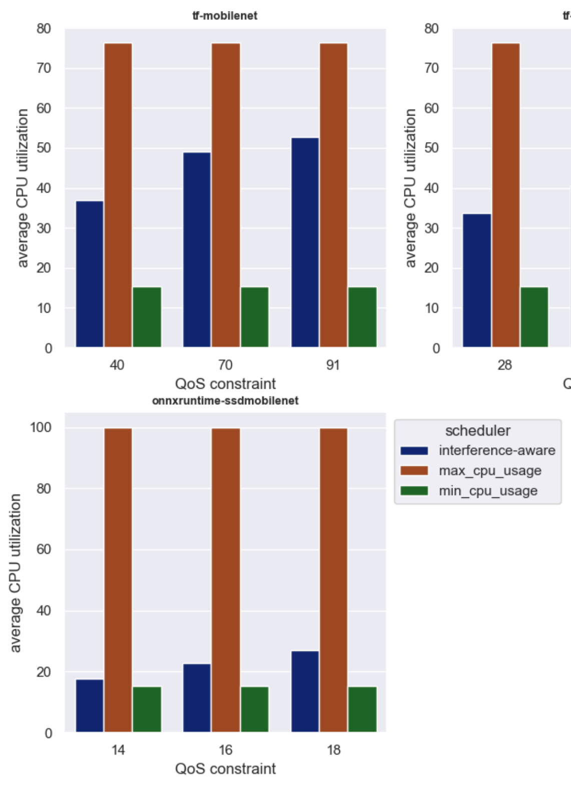
interference: 2



interference: 3



interference: 3



interference: 3

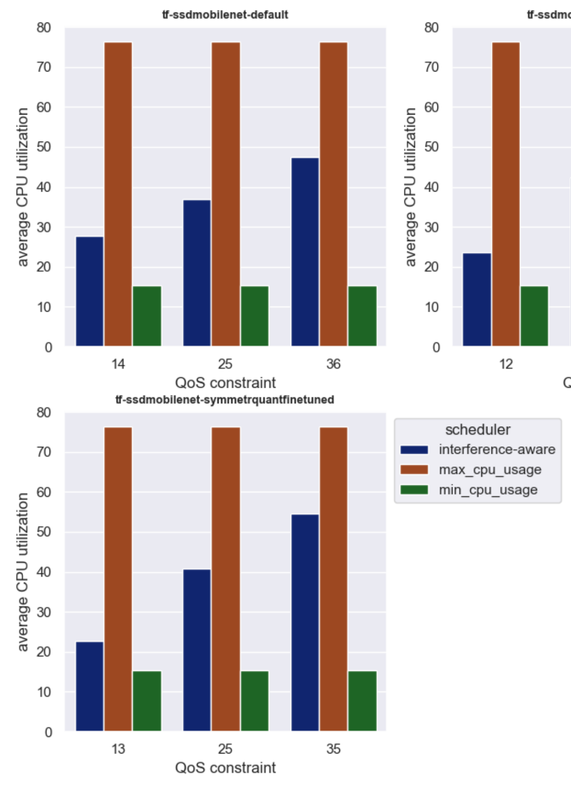


Figure 6.2: Average CPU utilization percentage comparison between the interference-aware scheduler, the max CPU usage scheduler and the min CPU usage scheduler. There are 3 different QoS constraints for each MLPerf Inference benchmark, and 3 different interference scenarios.

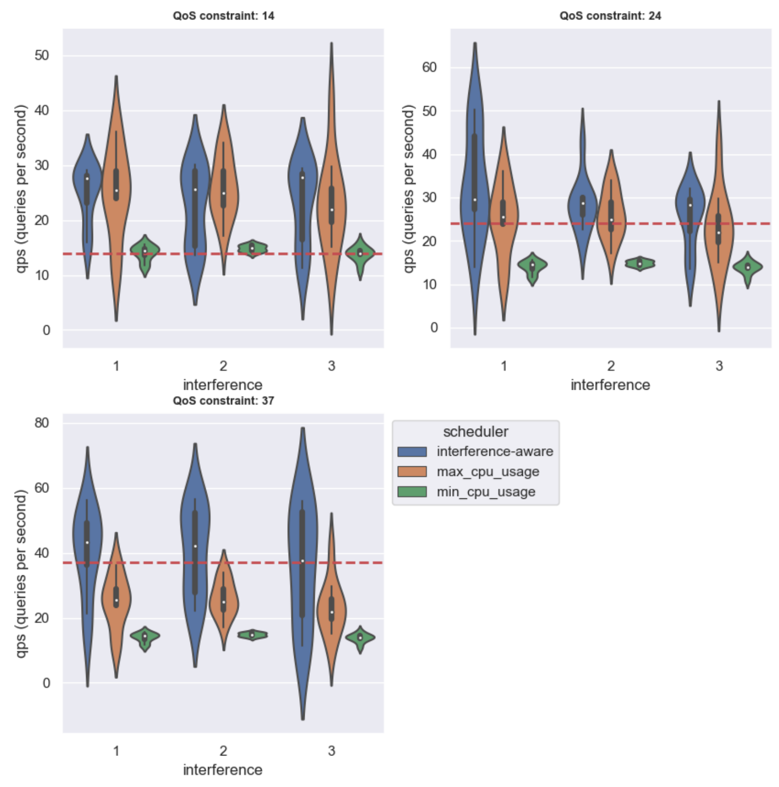
At a first glance at the figures, we see that our custom interference-aware scheduler manages to achieve an average CPU utilization somewhere in between what the min CPU usage and max CPU usage schedulers have. More specifically, our scheduler utilizes, on average, 2.4x more CPU than the min CPU usage scheduler and 2.3x less CPU than the max CPU usage scheduler. What that means is that, our custom scheduler is able to get close to or surpass the QoS constraint for the majority of the benchmarks' deployments, in contrast to the min CPU usage scheduler, while also doing so at a lower CPU utilization than the max CPU usage scheduler does.

In the benchmarks with the tensorflow backend, our custom scheduler utilizes significantly lower CPU at a low QoS constraint than the max CPU usage scheduler does. The difference becomes smaller the higher the QoS constraint gets, although our custom scheduler still manages to use 15-35% less CPU than the max CPU usage scheduler does, at the highest QoS constraint. As far as the benchmarks with the onnxruntime backend are concerned, our custom scheduler utilizes far less CPU in order to meet the target QoS than the max CPU usage scheduler does. More specifically, for the onnxruntime-resnet50 benchmark, our custom scheduler uses half or less CPU than the max CPU usage scheduler does, while being capable of presenting fewer QoS violations than both the min and max CPU usage schedulers do at middle and high QoS constraints. As for the mobilenet and ssd-mobilenet benchmarks with the onnxruntime backend, at an average of only about twice the CPU utilization of the min CPU usage scheduler, our custom scheduler manages to have significantly fewer QoS violations than both the min and max CPU usage schedulers have at middle and high QoS constraints.

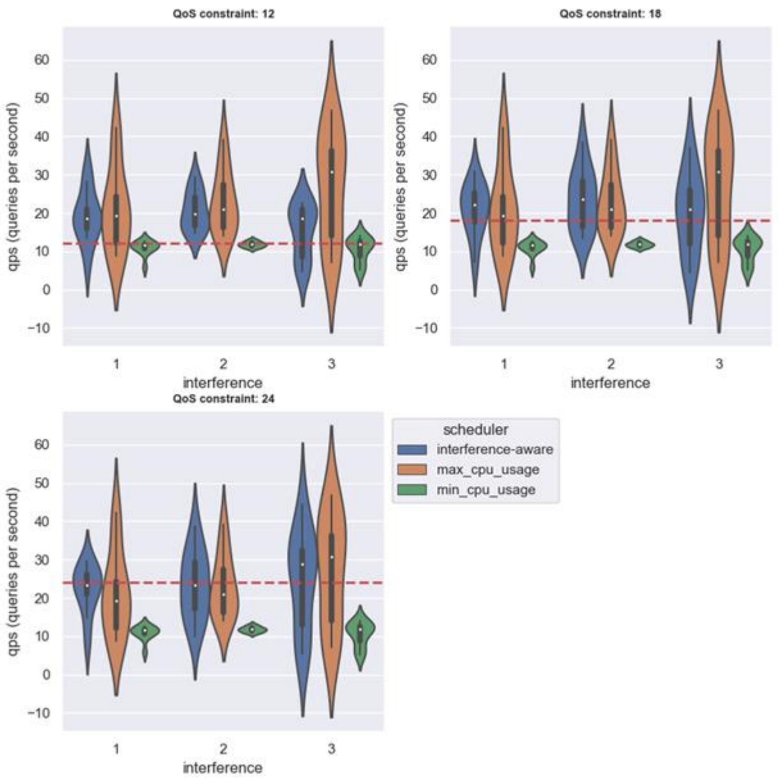
6.3.3 Performance Distribution

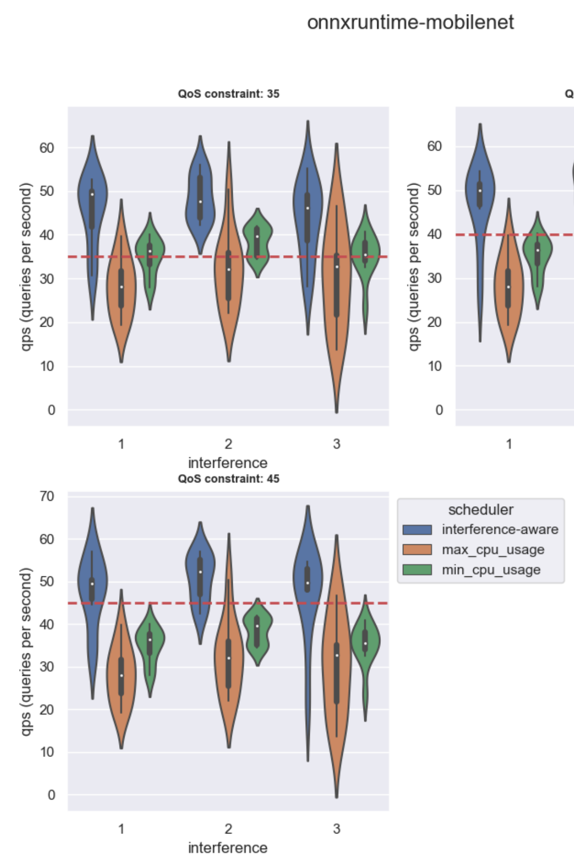
In this set of figures, we examine the QPS score each MLPerf Inference benchmark achieved in its 10 consecutive deployments served by each scheduler. We compare the performance distribution over these 10 deployments, between our custom interference-aware scheduler, the min CPU usage scheduler and the max CPU usage scheduler, with 3 different QoS constraints for each MLPerf Inference benchmark, and at 3 different interference scenarios.

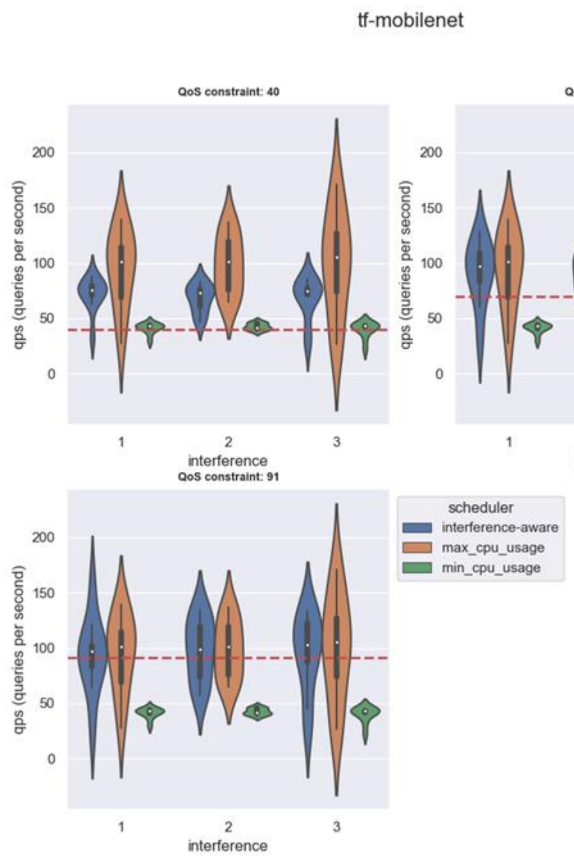
onnxruntime-resnet50



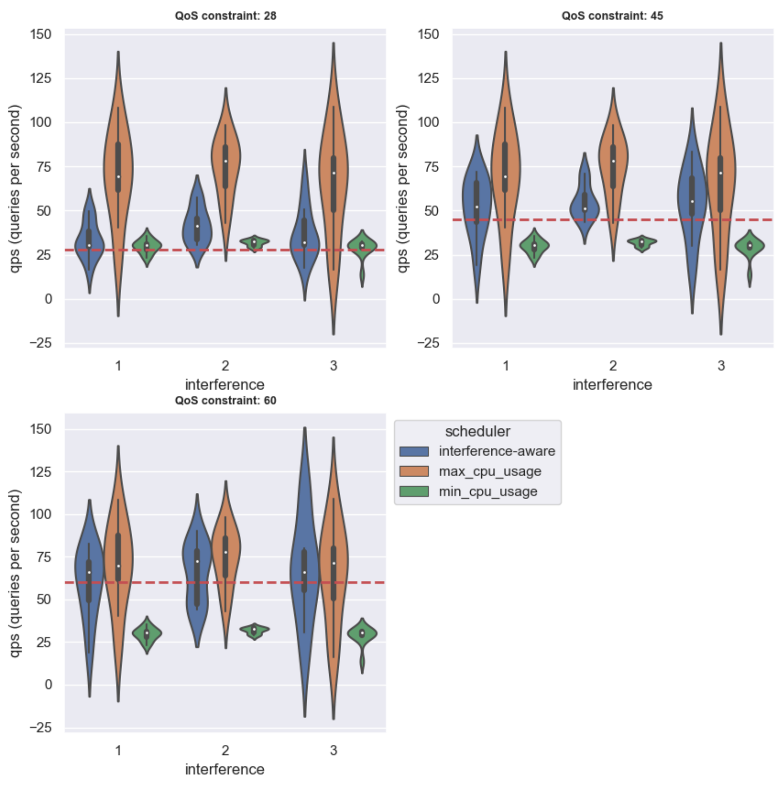
tf-resnet50



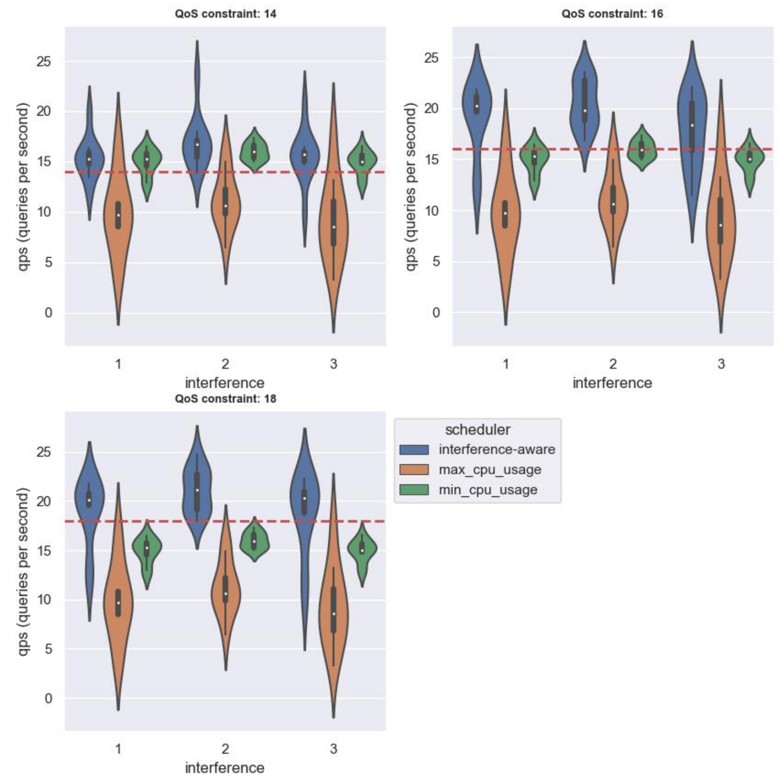

onnxruntime-mobilenet
QoS constraint: 35
QoS constraint: 40
QoS constraint: 45
qps (queries per second)
interference
scheduler
interference-aware
max_cpu_usage
min_cpu_usage

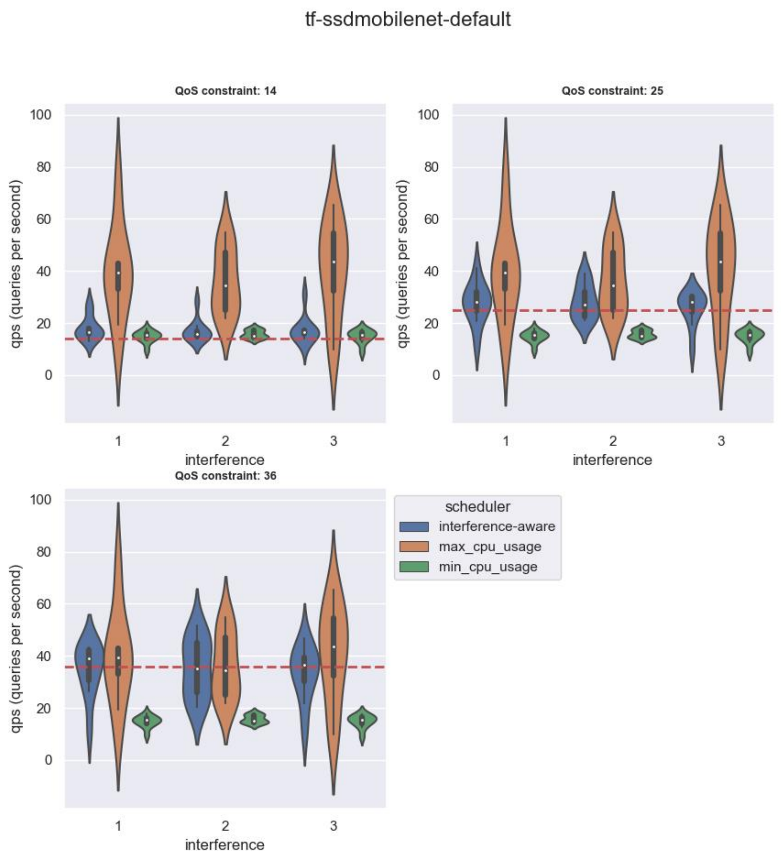

tf-mobilenet
QoS constraint: 40
QoS constraint: 70
QoS constraint: 91
qps (queries per second)
interference
scheduler
interference-aware
max_cpu_usage
min_cpu_usage

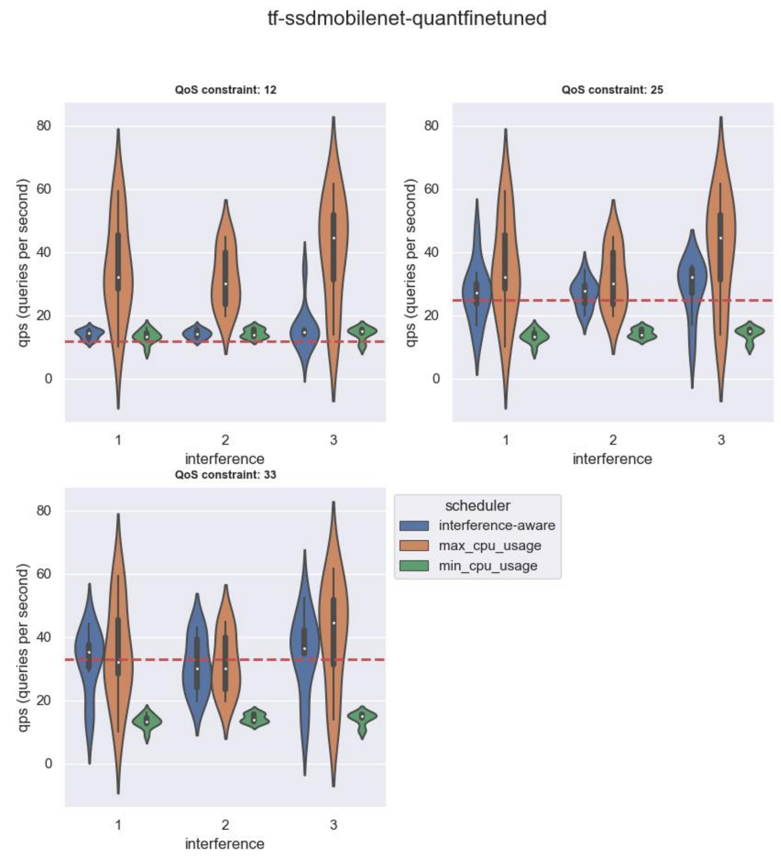
tf-mobilenet-quant



onnxruntime-ssdmobilenet




tf-ssdmobilenet-default
QoS constraint: 14
QoS constraint: 25
QoS constraint: 36
qps (queries per second)
interference
scheduler
interference-aware
max_cpu_usage
min_cpu_usage


tf-ssdmobilenet-quantfinetuned
QoS constraint: 12
QoS constraint: 25
QoS constraint: 33
qps (queries per second)
interference
scheduler
interference-aware
max_cpu_usage
min_cpu_usage

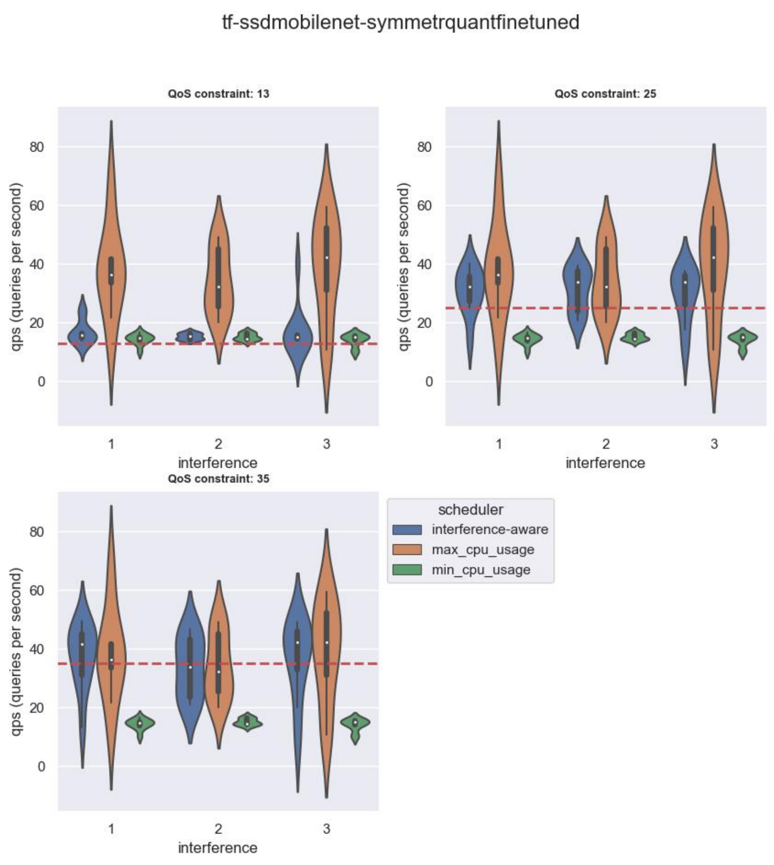


Figure 6.3: QPS distribution comparison between the interference-aware scheduler, the max CPU usage scheduler and the min CPU usage scheduler. There are 3 different QoS constraints for each MLPerf Inference benchmark, and 3 different interference scenarios.

What is readily noticeable from the figures, is that for all the MLPerf Inference benchmarks in all the interference scenarios, the median value of the performance distribution of our custom interference-aware scheduler stays remarkably close to or, in most cases, above the QoS constraint. In other words, our custom scheduler succeeds in meeting or surpassing the target QoS in the vast majority of situations, where the interference intensity fluctuates, making it an excellent choice for applications where an expected service quality is required.

Another interesting observation is that, in the cases of the benchmarks with the tensorflow backend, at a low QoS constraint, both our custom scheduler and the max CPU usage scheduler manage to stay above the QoS line. However, while our scheduler utilizes as much power from the CPU as required to guarantee that it will meet the target QoS, the max CPU usage scheduler overcompensates by utilizing all of the CPU and, thus, reaching far above the QoS constraint with no reason. At a mid and high QoS constraint, the performance difference between these two schedulers is lower, but still our custom scheduler manages to have a smaller interquartile range than the max CPU usage scheduler does, meaning most of the QPS scores it achieves are closer together, making it a more robust solution. It is also clear that the min CPU usage scheduler cannot reach the target QoS at all if it is at a middle value or higher.

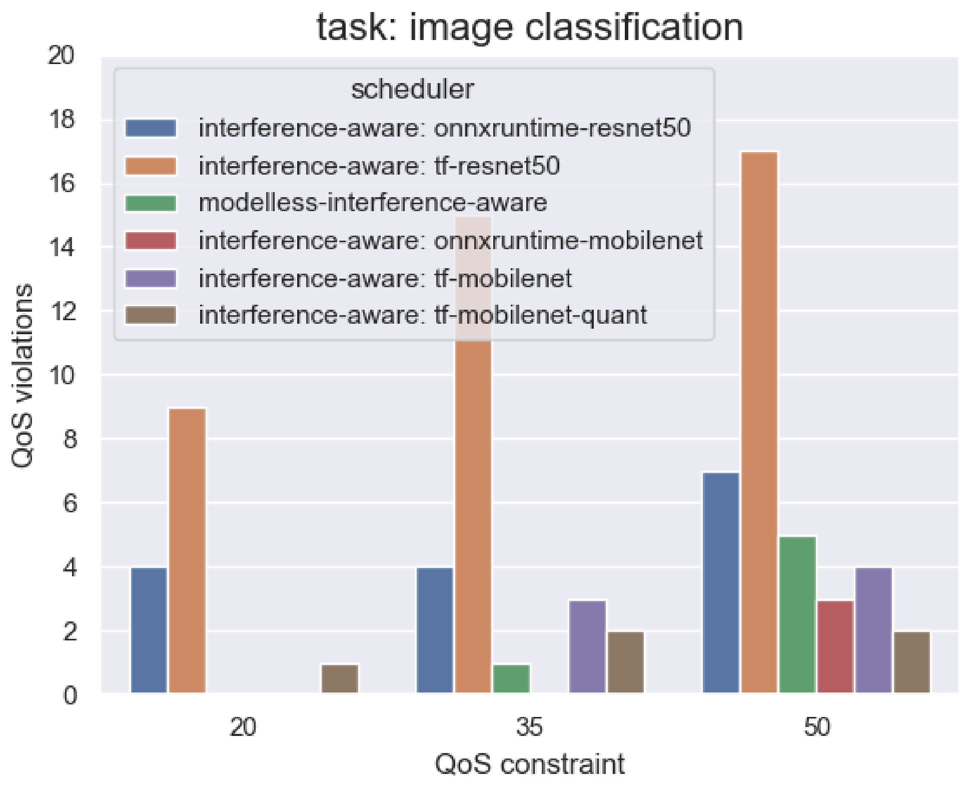
As far as the renset50 benchmark with the onnxruntime backend is concerned, at a low and mid QoS constraint, our custom scheduler and the max CPU usage scheduler perform similarly well, hitting the target QoS most of the time, while the min CPU usage scheduler only manages to do that at the low QoS value. At a high QoS constraint though, only our custom scheduler manages to reach the required QoS, with the other two schedulers performing poorly.

In the onnxruntime-mobilenet and onnxruntime-ssdmobilenet benchmarks, our custom scheduler presents a performance distribution with the highest probability of QPS scores landing above the target QoS, for all QoS constraints. In contrast, the min CPU usage scheduler only manages to reliably hit the QoS target at a low QoS constraint, and the higher the QoS value gets, the worse its performance. As for the max CPU usage scheduler, both its median value, as well as its interquartile value stay below the target QoS, at all QoS constraints.

6.4 Model-less Inference Engine Scheduler Results & Scheduler Comparison

6.4.1 Violations of the QoS constraint

In the first set of figures, we examine the number of times a scheduler did not meet the desired target QoS, but rather it scheduled an MLPerf Inference benchmark with a parallelism level that scored a lower QPS than the QoS constraint. We compare the number of QoS violations between our custom, model-less interference-aware scheduler (by specifying only what ML task the inference engines should perform) and our custom, model-specific interference-aware scheduler (that serves only a specific MLPerf Inference benchmark that performs that ML task), at 3 different QoS constraints at each ML task, in a single interference scenario.



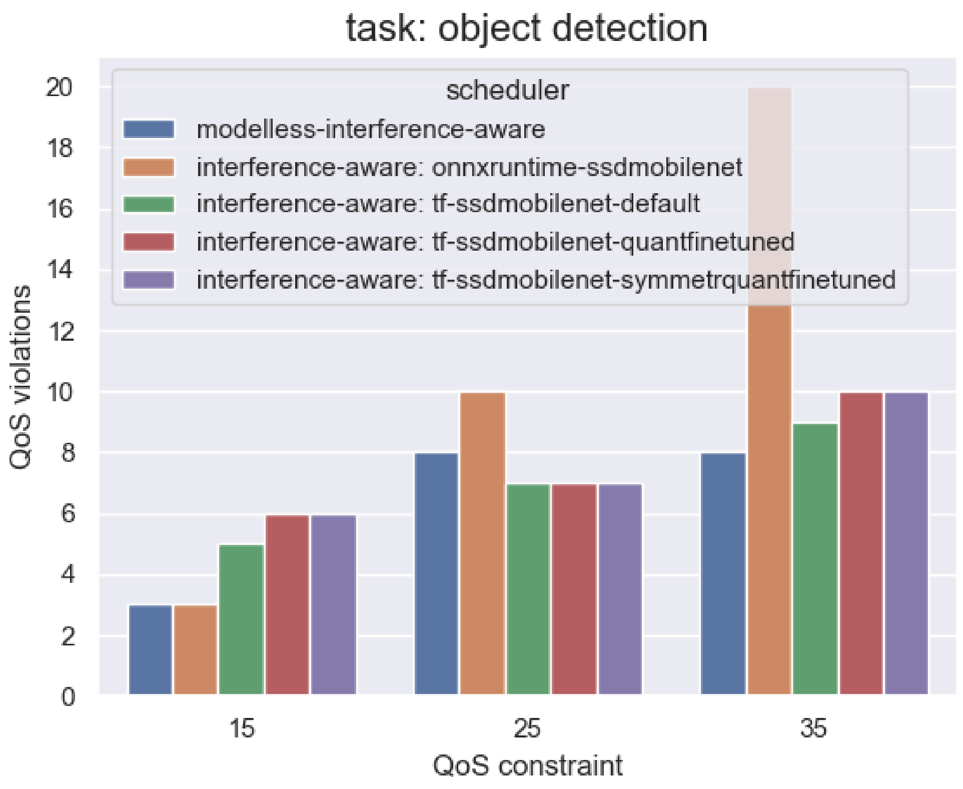


Figure 6.4: Number of QoS violations comparison between the model-specific interference-aware scheduler and the model-less interference-aware scheduler, at 3 different QoS constraints for each ML task.

Beginning the analysis with the image classification task, we clearly see that the best performers of all the MLPerf Inference benchmarks are the mobilenet with either an onnxruntime or tensorflow backend, and the quantized mobilenet with the tensorflow backend. The resnet50 benchmark, no matter the backend, violates all the QoS constraints at a higher rate than the rest of the benchmarks do each time.

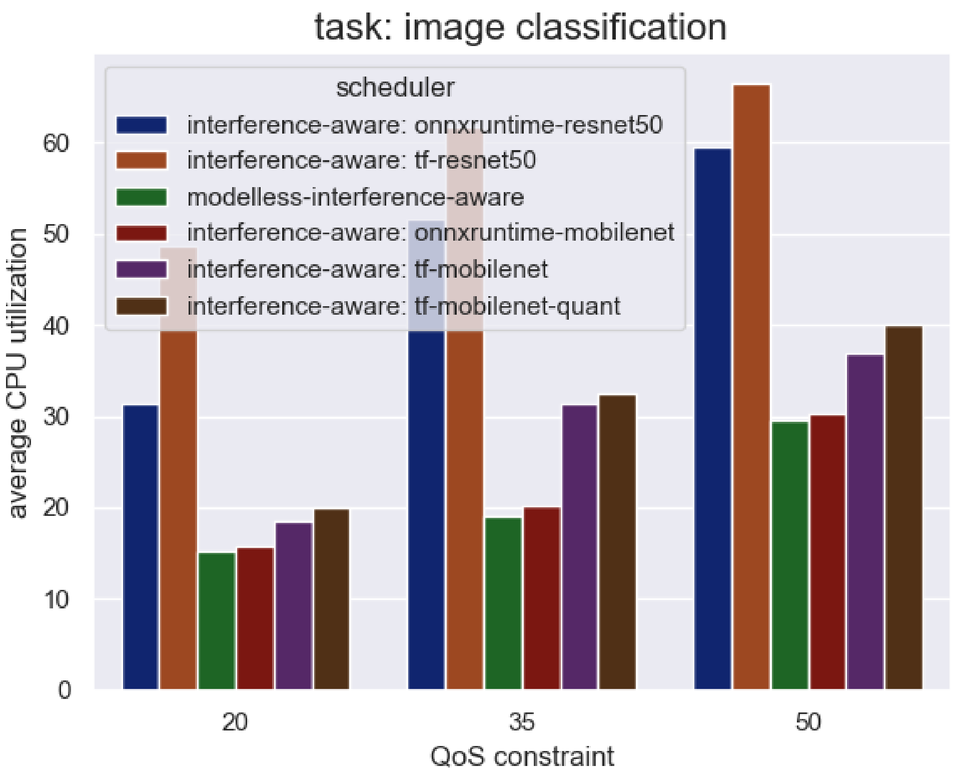
At a low target QoS, the model-less scheduler always uses the onnxruntime-mobilenet benchmark, that never violates the constraint. A similar impressive performance is seen by all of the aforementioned image classification best performers. When the QoS constraint is at an intermediate value, our model-less scheduler uses the onnxruntime-mobilenet benchmark most of the time, with a few changes to the tf-mobilenet. The model-less scheduler and the scheduler with the onnxruntime-mobilenet show the least QoS violations.

At the high QoS constraint, the model-less scheduler uses the tf-mobilenet benchmark most of the time, which has a higher QPS capability at a lower CPU utilization, and changes to the onnxruntime-mobilenet for a some of the deployments. However, due to the varying intensity of the interference conditions, the model-less scheduler relies on the QPS predictions of the corresponding trained ML Regression model for each MLPerf Inference benchmark, in order to choose the one with a high enough QPS prediction and the lowest CPU utilization. Because the QPS predictions of the Regression model are not perfect and, especially as the interference intensity increases, their values can be remarkably close for the best performing benchmarks and their parallelism levels, the inference engine substitutes, the model-less scheduler does, could, sometimes, result in a choice with a lower real QPS score. Consequently, as is the case in this interference scenario here, the model-less scheduler happens to violate the QoS constraint slightly more often than all the three best performing benchmarks, although it still performs well.

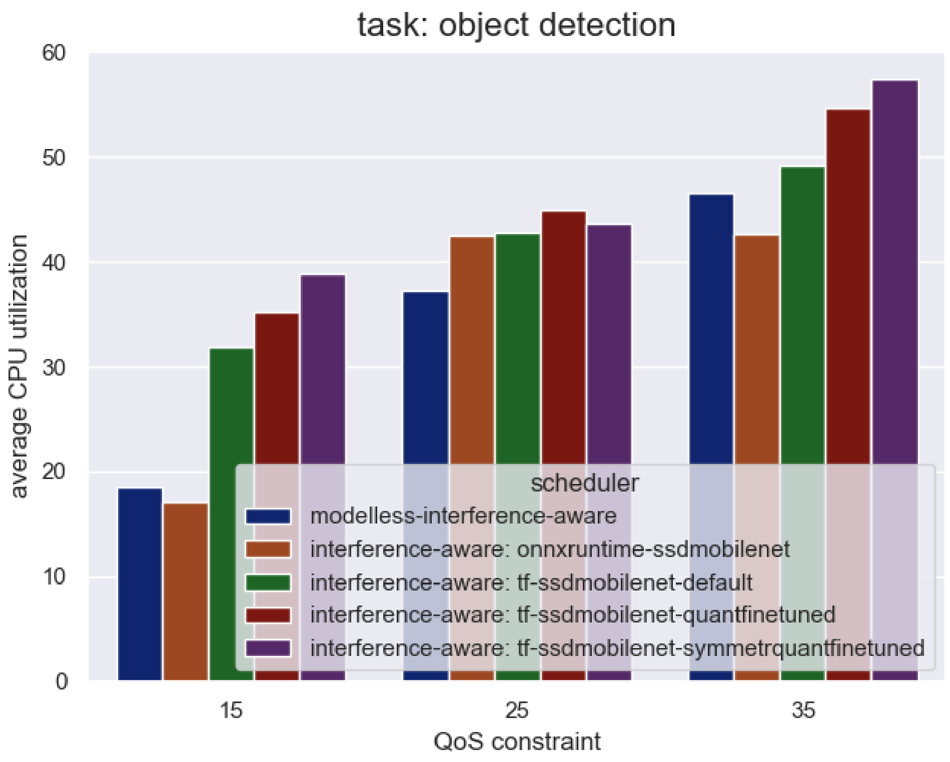
As far as the object detection task is concerned, our model-less scheduler mostly uses the onnxruntime-ssdmobilenet benchmark on a low QoS constraint, resulting in a similar satisfactory performance to the model-specific scheduler that only uses the onnxruntime-ssdmobilenet, that meets the target QoS almost every time. We notice that the schedulers that use the other MLPerf Inference benchmarks violate the low QoS constraint nearly twice as often as the model-less scheduler does. Interestingly, for the mid and high QoS constraints, the model-less scheduler turns to the tf-ssdmobilenet benchmark for most of the deployments. As a result, it has around the same QoS violations, on average, with the model-specific scheduler that serves the tf-ssdmobilenet, but it performs significantly better than the scheduler with the onnxruntime-ssdmobilenet benchmark does. The schedulers that use the last two of these MLPerf Inference benchmarks have a slightly worse performance than our model-less scheduler in the high QoS constraint.

6.4.2 Resources Utilization

In the next set of figures, we examine what percentage of the resources of the cluster are offered to the deployed MLPerf Inference benchmarks each time by the schedulers. More specifically, we compare the average CPU utilization percentage of the 20 consecutive deployments of inference engines on the cluster, between our custom model-specific interference-aware scheduler, and our custom, model-less interference-aware scheduler, with 3 different QoS constraints for each ML task, at a single interference scenario.



task: image classification
scheduler
interference-aware: onnxruntime-resnet50
interference-aware: tf-resnet50
modelless-interference-aware
interference-aware: onnxruntime-mobilenet
interference-aware: tf-mobilenet
interference-aware: tf-mobilenet-quant
average CPU utilization
0
10
20
30
40
50
60
20
35
50
QoS constraint



task: object detection
scheduler
modelless-interference-aware
interference-aware: onnxruntime-ssdmobilenet
interference-aware: tf-ssdmobilenet-default
interference-aware: tf-ssdmobilenet-quantfinetuned
interference-aware: tf-ssdmobilenet-symmetrquantfinetuned
average CPU utilization
0
10
20
30
40
50
60
15
25
35
QoS constraint

Figure 6.5: Average CPU utilization percentage comparison between the model-specific interference-aware scheduler, and the model-less interference-aware scheduler, at 3 different QoS constraints for each ML task.

For the image classification task, the model-less interference-aware scheduler manages to keep the average CPU utilization, at a minimum, across all QoS constraints, slightly lower than what the onnxruntime-mobilenet benchmark of the model-specific scheduler achieves. We note that the least CPU intensive benchmark, as served from the model-specific scheduler, is the onnxruntime-mobilenet, with the tf-mobilenet and quantized tf-mobilenet falling behind. The worst performer in all QoS constraints is, again, the resnet50 benchmark, with either of the backends, which has the highest CPU utilization and, as we observed in the previous section, the higher amount of QoS violations.

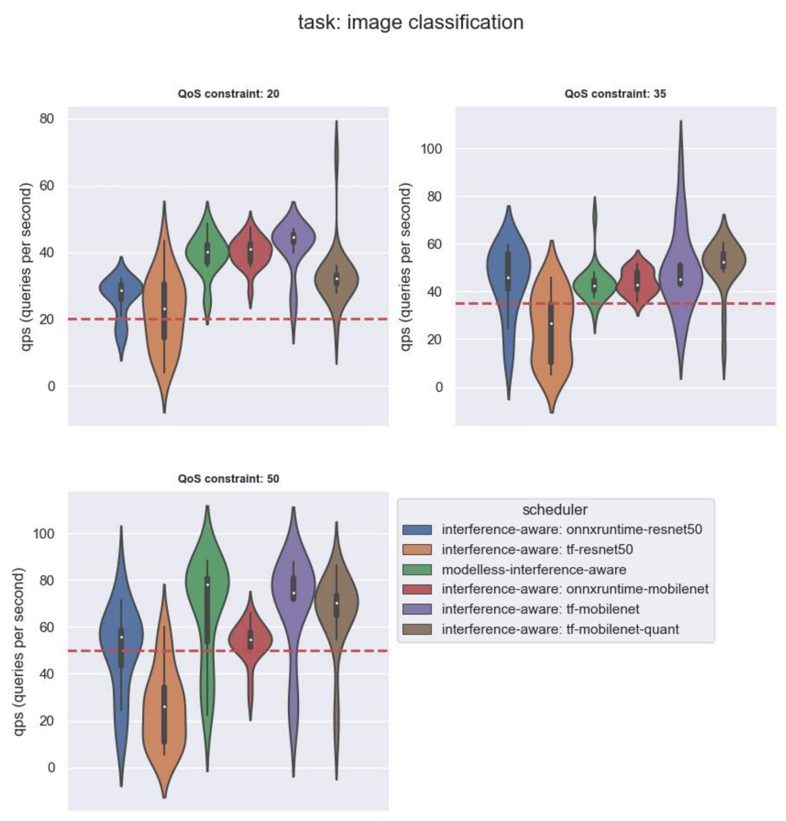
At the lowest QoS constraint, most of the benchmarks score a low CPU utilization value, as they can reach that QPS with low resource usage. At the mid and high QoS constraints though, the difference between the CPU utilization of the benchmarks grows bigger, with the model-specific scheduler serving the onnxruntime-mobilenet benchmark, and the model-less scheduler with the onnxruntime-mobilenet and tf-mobilenet substitutions throughout the experiment, achieving the lowest violations of the target QoS.

As far as the object detection task of the MLPerf Inference benchmarks is concerned, the model-less scheduler has, on average, the same CPU utilization, at all QoS constraints, as the model-specific scheduler with the onnxruntime-ssdmobilenet benchmark, which is the least resource intensive benchmark from this benchmark group. We should recall here that, despite their similar CPU utilization, the model-less scheduler greatly outperforms the onnxruntime-ssdmobilenet at the QoS violations at the mid and high QoS constraints, since it selects the tf-ssdmobilenet then, most of the time.

We observe from the figure that, at the low value of the QoS constraint, the model-less scheduler and the onnxruntime-ssdmobilenet use nearly half the CPU of the system as the rest of the benchmarks do, while also scoring the fewest QoS violations. At the mid and high QoS constraints, the model-less scheduler still performs better resource utilization-wise than the tf-ssdmobilenet and its tensorflow variants, while having around the same QoS violations as them.

6.4.3 Performance Distribution

In this set of figures, we examine the QPS score each scheduler achieved in the 20 consecutive inference engine deployments. We compare the performance distribution over these 20 deployments, between our custom, model-specific interference-aware scheduler and our custom, model-less interference-aware scheduler, at 3 different QoS constraints for each ML task, in a single interference scenario.


task: image classification
QoS constraint: 20
QoS constraint: 35
QoS constraint: 50
qps (queries per second)
scheduler
interference-aware: onnxruntime-resnet50
interference-aware: tf-resnet50
modelless-interference-aware
interference-aware: onnxruntime-mobilenet
interference-aware: tf-mobilenet
interference-aware: tf-mobilenet-quant

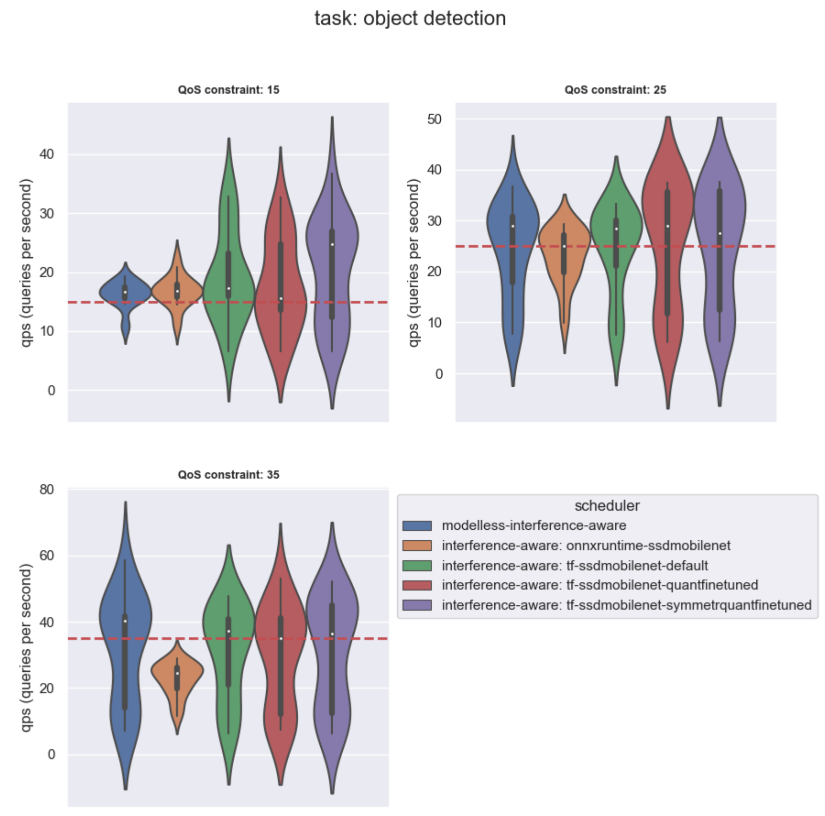

task: object detection
QoS constraint: 15
QoS constraint: 25
QoS constraint: 35
qps (queries per second)
scheduler
modelless-interference-aware
interference-aware: onnxruntime-ssdmobilenet
interference-aware: tf-ssdmobilenet-default
interference-aware: tf-ssdmobilenet-quantfinetuned
interference-aware: tf-ssdmobilenet-symmetrquantfinetuned

Figure 6.6: QPS distribution comparison between the model-specific interference-aware scheduler and the model-less interference-aware scheduler, at 3 different QoS constraints for each ML task.

Beginning our observations from the image classification figure, the model-less scheduler manages to contain its QPS scores relatively close to the low and mid QoS constraints, where its performance distribution is quite similar to the onnxruntime-mobilenet at both these cases. At these lower values of QoS, the other benchmarks present a wider QPS distribution. It is obvious that, at all the QoS constraints, the resnet50 with tensorflow backend drifts its interquartile range considerably low, resulting in an increased number of QoS violations. What is more, the onnxruntime-resnet50, while not as bad, still has a distribution that leans into lower levels than the QoS constraint.

At the highest QoS constraint, the model-less scheduler shows more performance variability than either the onnxruntime-mobilenet, the tf-mobilenet or the quantized tf-mobilenet, due to some unfruitful decisions on inference engine changes, which costs it a few QoS violations. However, as we see from the figure, the interquartile range of its QPS distribution still remains above the QoS constraint, resulting in a good overall performance. The onnxruntime-mobilenet is the more robust performer overall, and especially at the high QoS constraint.

Shifting our observations to the object detection task, we see that the model-less scheduler has a very concentrated QPS distribution at the low QoS constraint, similar to the model-specific scheduler that serves the onnxruntime-ssdmobilenet benchmark, while the tensorflow ssd-mobilenet variants have far more disperse QPS values. As the QoS constraint increases, the model-less scheduler widens its performance distribution by changing almost exclusively to the tf-ssdmobilenet benchmark, something that allows it to keep up with the QoS constraint, as opposed to the onnxruntime-ssdmobilenet, which eventually completely misses the target QoS at the high value.

Chapter 7

Conclusion

In this thesis, we designed an interference and resource aware, predictive scheduling framework, for ML inference engines, in order to meet application-specific QoS constraints, while being as little resource-intensive to the system as possible. We also extended our design to a model-less scheduling framework to deal with the large number of diverse inference engines. We evaluated our proposed scheduling framework, by using a set of inference engines from the MLPerf Inference Benchmark Suite on Kubernetes system, at different interference scenarios, generated with the use of the iBench workload suite. We showed that for a set of different required QoS constraints, our scheduling framework violates QoS constraints, on average, less often, with a more robust performance around the target QoS, and with a moderate amount of CPU resource utilization depending on the target QoS and resource load, compared to the min and max CPU utilization inference serving system. Finally, we showed that our model-less approach further improves the average number of violations of the QoS constraints, and the average CPU utilization of our scheduling framework.

Bibliography

[1] Romero, Francisco, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. "{INFaaS}: Automated Model-less Inference Serving." In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pp. 397-411. 2021.

[2] Gujarati, Arpan, Sameh Elnikety, Yuxiong He, Kathryn S. McKinley, and Björn 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.

[3] Wang, Luping, Lingyun Yang, Yinghao Yu, Wei Wang, Bo Li, Xianchao Sun, Jian He, and Liping 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.

[4] Choi, Seungbeom, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. "Serving Heterogeneous Machine Learning Models on {Multi-GPU} Servers with {Spatio-Temporal} Sharing." In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pp. 199-216. 2022.

[5] Gao, Wei, Qinghao Hu, Zhisheng Ye, Peng Sun, Xiaolin Wang, Yingwei Luo, Tianwei Zhang, and Yonggang Wen. "Deep Learning Workload Scheduling in GPU Datacenters: Taxonomy, Challenges and Vision." *arXiv preprint arXiv:2205.11913* (2022).

[6] "Docker Official Website." [Online]. Available: https://www.docker.com/

[7] "Kubernetes (Official Website)." [Online]. Available: https://kubernetes.io/

[8] Reddi, Vijay Janapa, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson et al. "Mlperf inference benchmark." In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 446-459. IEEE, 2020.

[9] "MLPerf™ Inference Benchmark Suite for Image Classification and Object Detection Tasks [Online]. Available: https://github.com/mlcommons/inference/tree/master/vision/classifcation_and_detection

[10] "MLPerf Inference Load Generator." [Online]. Available: https://github.com/mlcommons/inference/tree/master/loadgen

[11] Delimitrou, Christina, and Christos Kozyrakis. "ibench: Quantifying interference for datacenter applications." In *2013 IEEE international symposium on workload characterization (IISWC)*, pp. 23-33. IEEE, 2013.

[12] Mitchell, Tom, Bruce Buchanan, Gerald DeJong, Thomas Dietterich, Paul Rosenbloom, and Alex Waibel. "Machine learning." *Annual review of computer science* 4, no. 1 (1990): 417-433.

[13] Samuel, Arthur L. "Some studies in machine learning using the game of checkers. II—recent progress." *Computer Games I* (1988): 366-400.

[14] Lindsay, Richard P. "The Impact of Automation on Public Administration." *Western Political Quarterly* 17, no. 3 (1964): 78-81.

[15] Koza, John R., Forrest H. Bennett, David Andre, and Martin A. Keane. "Automated design of both the topology and sizing of analog electrical circuits using genetic programming." In *Artificial intelligence in design'96*, pp. 151-170. Springer, Dordrecht, 1996.

[16] Hu, Junyan, Hanlin Niu, Joaquin Carrasco, Barry Lennox, and Farshad Arvin. "Voronoi-based multi-robot autonomous exploration in unknown environments via deep reinforcement learning." *IEEE Transactions on Vehicular Technology* 69, no. 12 (2020): 14413-14423.

[17] Russell, Stuart J. *Artificial intelligence a modern approach*. Pearson Education, Inc., 2010.

[18] Bishop, Christopher M., and Nasser M. Nasrabadi. *Pattern recognition and machine learning*. Vol. 4, no. 4. New York: springer, 2006.

[19] Hilt, Donald E., and Donald W. Seegrist. *Ridge, a computer program for calculating ridge regression estimates*. Department of Agriculture, Forest Service, Northeastern Forest Experiment Station, 1977.

[20] Sra, Suvrit, Sebastian Nowozin, and Stephen J. Wright, eds. *Optimization for machine learning*. Mit Press, 2012.

[21] Cortes, Corinna, and Vladimir Vapnik. "Support-vector networks." *Machine learning* 20, no. 3 (1995): 273-297.

[22] Drucker, Harris, Christopher J. Burges, Linda Kaufman, Alex Smola, and Vladimir Vapnik. "Support vector regression machines." *Advances in neural information processing systems* 9 (1996).

[23] Cover, Thomas, and Peter Hart. "Nearest neighbor pattern classification." *IEEE transactions on information theory* 13, no. 1 (1967): 21-27.

[24] Hastie, Trevor, Robert Tibshirani, Jerome H. Friedman, and Jerome H. Friedman. *The elements of statistical learning: data mining, inference, and prediction*. Vol. 2. New York: springer, 2009.

[25] Rosenblatt, Frank. *Principles of neurodynamics. perceptrons and the theory of brain mechanisms*. Cornell Aeronautical Lab Inc Buffalo NY, 1961.