ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Edge-to-Cloud Synergy for Architecture-Driven High-Performance Orchestration for AI Inference

Σταθοπούλου Φωτεινή

Επιβλέπων : Δημήτριος Ι. Σούντρης
Καθηγητής ΕΜΠ

Αθήνα, Ιούλιος 2025

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Edge to-Cloud Synergy for Architecture Driven High-Performance Orchestration for AI Inference

Σταθοπούλου Φωτεινή

Επιβλέπων : Δημήτριος Ι. Σούντρης
Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 3η Ιουλίου 2025.

Δημήτριος Σούντρης	Σωτήρης Ξύδης	Ρουσσάκη Ιωάννα
(Υπογραφή)	(Υπογραφή)	(Υπογραφή)
.....	.....	.....
Καθηγητής ΕΜΠ	Καθηγητής ΕΜΠ	Αναπληρώτρια Καθηγήτρια ΕΜΠ

Αθήνα, Ιούλιος 2025

(Υπογραφή)

.....

Σταθοπούλου Φωτεινή

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Η ταχεία εξέλιξη της Τεχνητής Νοημοσύνης (AI) και της Μηχανικής Μάθησης (ML) έχει αυξήσει σημαντικά τις υπολογιστικές απαιτήσεις, ιδιαίτερα για φόρτους εργασίας εξυπηρέτησης προβλέψεων. Ενώ οι παραδοσιακές υλοποιήσεις στο νέφος προσφέρουν κλιμάκωση, αντιμετωπίζουν προκλήσεις όπως η συμφόρηση δικτύου, η υψηλή κατανάλωση ενέργειας και οι προβληματισμοί σχετικά με την ιδιωτικότητα. Αντιθέτως, το edge computing παρέχει βιώσιμες εναλλακτικές λύσεις χαμηλής καθυστέρησης, αλλά δεσμεύεται από τους περιορισμένους υπολογιστικούς πόρους. Σε αυτή τη διπλωματική, παρουσιάζουμε το SynergAI, ένα καινοτόμο πλαίσιο σχεδιασμένο για εξυπηρέτηση προβλέψεων με επίγνωση επιδόσεων και αρχιτεκτονικής σε ετερογενείς υποδομές edge-to-cloud. Βασισμένο σε έναν ολοκληρωμένο χαρακτηρισμό επιδόσεων σύγχρονων μηχανών πρόβλεψης, το SynergAI ενσωματώνει έναν συνδυασμό πολιτικών λήψης αποφάσεων εκτός σύνδεσης και σε πραγματικό χρόνο για την παροχή έξυπνης, ελαφριάς και αρχιτεκτονικά ενήμερης δρομολόγησης. Με τη δυναμική κατανομή φόρτων εργασίας σε διάφορες αρχιτεκτονικές υλικού, ελαχιστοποιεί αποτελεσματικά τις παραβιάσεις Ποιότητας Υπηρεσίας (QoS). Υλοποιούμε το SynergAI σε ένα οικοσύστημα βασισμένο στο Kubernetes και αξιολογούμε την αποδοτικότητά του. Τα αποτελέσματά μας δείχνουν ότι η αρχιτεκτονικά καθοδηγούμενη εξυπηρέτηση προβλέψεων επιτρέπει βελτιστοποιημένες και αρχιτεκτονικά ενήμερες υλοποιήσεις σε αναδυόμενες πλατφόρμες υλικού, επιτυγχάνοντας μια μέση μείωση 2.4× στις παραβιάσεις Ποιότητας Υπηρεσίας σε σύγκριση με μια λύση State-of-the-Art (SotA).

Λέξεις Κλειδιά— Νέφος, Edge, Πρόβλεψη, Δρομολόγηση, Επίγνωση Επιδόσεων, Επίγνωση Αρχιτεκτονικής

Abstract

The rapid evolution of Artificial Intelligence (AI) and Machine Learning (ML) has significantly heightened computational demands, particularly for inference-serving workloads. While traditional cloud-based deployments offer scalability, they face challenges such as network congestion, high energy consumption, and privacy concerns. In contrast, edge computing provides low-latency and sustainable alternatives but is constrained by limited computational resources. In this thesis, we introduce `SynergAI`, a novel framework designed for performance- and architecture-aware inference serving across heterogeneous edge-to-cloud infrastructures. Built upon a comprehensive performance characterization of modern inference engines, `SynergAI` integrates a combination of offline and online decision-making policies to deliver intelligent, lightweight, and architecture-aware scheduling. By dynamically allocating workloads across diverse hardware architectures, it effectively minimizes Quality of Service (QoS) violations. We implement `SynergAI` within a Kubernetes-based ecosystem and evaluate its efficiency. Our results demonstrate that architecture-driven inference serving enables optimized and architecture-aware deployments on emerging hardware platforms, achieving an average reduction of 2.4× in QoS violations compared to a State-of-the-Art (SotA) solution.

Keywords— Cloud, Edge, Inference, Scheduling, Performance-aware, Architecture-aware

Ευχαριστίες

Αρχικά, θα ήθελα να ευχαριστήσω τον επιβλέποντά μου, Καθηγητή Δημήτριο Σούντρη ΕΜΠ, ο οποίος μου έδωσε την ευκαιρία να εκπονήσω τη διπλωματική μου εργασία στο Εργαστήριο Μικροϋπολογιστών και Ψηφιακών Συστημάτων (MicroLab) στο ΕΜΠ. Επίσης, θα ήθελα να εκφράσω τις θερμές μου ευχαριστίες στον υποψήφιο διδάκτορα Άγγελο Φερίκογλου και στον Δρ. Μανώλη Κατσαραγάκη για την πολύτιμη καθοδήγηση και τη συνεργασία τους καθ΄ όλη τη διάρκεια της διπλωματικής μου εργασίας. Τέλος, θα ήθελα να ευχαριστήσω την οικογένειά μου και τους φίλους μου για την υποστηρίξη τους κατά τη διάρκεια των σπουδών μου.

Contents

List of Figures

List of Tables

Chapter 1

Εκτεταμένη Ελληνική Περίληψη

1 Εισαγωγή

Τα τελευταία χρόνια, η ταχεία πρόοδος των εφαρμογών AI και ML και η εκτεταμένη ενσωμάτωσή τους στην καθημερινή ζωή έχει φέρει τη νέα εποχή των ευφυών συστημάτων πληροφορικής. Σύγχρονα μοντέλα βαθιάς μάθησης όπως το GPT-3 [2] και το DeepSeek-V2 [3], με 175 και 236 δισεκατομμύρια παραμέτρους αντίστοιχα, επιβάλλουν σημαντικές απαιτήσεις πόρων και κινδυνεύουν να προκαλέσουν σημεία συμφόρησης στην απόδοση.

Παραδοσιακά, αυτά τα μοντέλα διέπονται από απαιτήσεις QoS [4] καθώς και από Στόχους Επιπέδου Υπηρεσιών (SLOs) ή Συμφωνίες Επιπέδου Υπηρεσιών (SLAs) [5]. Για την κάλυψη αυτών των αναγκών, τα μοντέλα συνήθως αναπτύσσονται σε πλατφόρμες Cloud computing υψηλής απόδοσης, οι οποίες βοηθούν στην άμβλυνση των σημείων συμφόρησης επιδόσεων ενώ προσφέρουν κλιμακούμενες, ευέλικτες και οικονομικά αποδοτικές λύσεις. Τα σύγχρονα συστήματα Cloud ενισχύουν τους φόρτους εργασίας ML/DL χρησιμοποιώντας κατανεμημένη υπολογιστική, εξειδικευμένους επιταχυντές υλικού (π.χ., GPUs, TPUs)[6], και βελτιστοποιημένη δικτύωση. Πολλοί πάροχοι προσφέρουν Machine Learning as a Service (MLaaS), παρέχοντας ολοκληρωμένες λύσεις για την εκπαίδευση, την ανάπτυξη και τη πρόβλεψη μοντέλων. Αξιοσημείωτες πλατφόρμες MLaaS περιλαμβάνουν το Google Cloud Vertex AI [7], το AWS SageMaker [8], το Microsoft Azure ML [9] και το IBM Watson ML [10]. Ωστόσο, καθώς οι απαιτήσεις αυξάνονται, η συμφόρηση του Cloud γίνεται μεγαλύτερη πρόκληση, ενώ η απόσταση των κέντρων δεδομένων από τις συσκευές των χρηστών προκαλεί μεγάλους χρόνους μεταφοράς και υποβαθμισμένη ποιότητα υπηρεσίας.

Για την αντιμετώπιση αυτών των περιορισμών, μέρος του υπολογιστικού φορτίου μετατοπίζεται στο Edge, φέρνοντας την επεξεργασία πιο κοντά στο σημείο παραγωγής των δεδομένων. Εξειδικευμένο υλικό AI και βελτιστοποιημένα πλαίσια ML επιτρέπουν αποδοτική πρόβλεψη απευθείας σε συσκευές Edge, προσφέροντας εφαρμογές χαμηλής κατανάλωσης ενέργειας και υψηλής απόδοσης, και διαμορφώνοντας το σύγχρονο πρότυπο του Edge AI [11]. Παρόλα αυτά, το Edge

computing λειτουργεί με πιο περιορισμένους πόρους σε σύγκριση με το Cloud, καθιστώντας την αποτελεσματική συνέργεια Edge-to-Cloud καθοριστική. Μια σημαντική πρόκληση είναι η ενσωμάτωση των εγγυήσεων QoS του Cloud με την αποδοτικότητα του Edge, αντιμετωπίζοντας παράλληλα αρκετές βασικές προκλήσεις: **i) Χρονοπρογραμματισμός Edge-to-Cloud & Ετερογένεια:** Η αποτελεσματική συνεργασία μεταξύ Edge και Cloud εξαρτάται από τον ευφυή χρονοπρογραμματισμό των φόρτων εργασίας πρόβλεψης, λαμβάνοντας υπόψη τις διαφορετικές αρχιτεκτονικές επεξεργαστών, όπως ARM, RISC-V και x86. **ii) Ανάπτυξη Πρόβλεψης με Επίγνωση Αρχιτεκτονικής:** Πολλές πλατφόρμες Edge προσφέρουν ρυθμιζόμενες λειτουργίες ισχύος και απόδοσης που εισάγουν συμβιβασμούς μεταξύ ενεργειακής απόδοσης και ταχύτητας πρόβλεψης. Η βέλτιστη ανάπτυξη απαιτεί δυναμική ρύθμιση των παραμέτρων εκτέλεσης ώστε να ευθυγραμμίζονται με τις απαιτήσεις του φόρτου εργασίας. **iii) Διαφορετικές Απαιτήσεις QoS και SLA:** Διαφορετικοί τομείς εφαρμογών επιβάλλουν ποικίλους περιορισμούς QoS, από καθυστέρηση επιπέδου χιλιοστών του δευτερολέπτου έως μεγαλύτερη ανοχή σε μεταβλητότητα. **iv) Μεταβλητότητα Μηχανής Πρόβλεψης:** Το ευρύ φάσμα πλαισίων εξυπηρέτησης πρόβλεψης εισάγει πολυπλοκότητα στην επιλογή του βέλτιστου μοντέλου και backend για ανάπτυξη σε διαφορετικούς κόμβους υλικού. Για την αντιμετώπιση αυτών των προκλήσεων, προηγούμενες έρευνες έχουν διερευνήσει λύσεις εξυπηρέτησης πρόβλεψης για το Edge [12, 13], το Cloud [4, 14], και το ευρύτερο συνεχές Edge-to-Cloud [15, 16]. Ωστόσο, οι υπάρχουσες προσεγγίσεις στερούνται πλαισίων χρονοπρογραμματισμού με επίγνωση αρχιτεκτονικής που βελτιστοποιούν δυναμικά την εξυπηρέτηση πρόβλεψης, προσαρμόζοντας στις διαφορετικές δυνατότητες των ετερογενών κόμβων. Επιπλέον, ενώ έχουν προταθεί διάφορες τεχνικές χρονοπρογραμματισμού για μεμονωμένα επίπεδα (Edge ή Cloud) και σε όλο το συνεχές, συχνά δεν εκμεταλλεύονται πλήρως τη δυναμική προσαρμογή πόρων και τις λειτουργίες εκτέλεσης με γνώμονα την αρχιτεκτονική.

Σε αυτή την εργασία, παρουσιάζουμε το **SynergAI**, ένα καινοτόμο πλαίσιο σχεδιασμένο για αποδοτική εξυπηρέτηση προβλέψεων, προσαρμοσμένη στην αρχιτεκτονική, σε ετερογενείς κόμβους Edge-to-Cloud. Ο κύριος στόχος βελτιστοποίησης του SynergAI είναι η ελαχιστοποίηση των παραβιάσεων QoS για μηχανές πρόβλεψης που αναπτύσσονται σε όλο το συνεχές, διασφαλίζοντας αποδοτική τοποθέτηση μηχανών με γνώμονα την αρχιτεκτονική. Η λύση μας βασίζεται σε έναν ολοκληρωμένο χαρακτηρισμό και ανάλυση της απόδοσης και της αρχιτεκτονικής διακριτών μηχανών πρόβλεψης, σε διαφορετικούς κόμβους Edge/Cloud και καταστάσεις λειτουργίας. Συγκεκριμένα, εστιάζουμε στις δυνατότητες συμβιβασμού απόδοσης και πόρων των κόμβων Edge, επιτρέποντας την υλοποίηση βελτιστοποιημένων στρατηγικών ανάπτυξης, προσαρμοσμένων στα μοναδικά χαρακτηριστικά κάθε κόμβου. Το αποτέλεσμα αυτής της ανάλυσης παρέχει τις κρίσιμες παραμέτρους

συντονισμού για τον προτεινόμενο ενορχηστρωτή Edge-to-Cloud. Το SynergAI χρησιμοποιεί ένα πλαίσιο χρονοπρογραμματισμού, με επίγνωση του QoS και της αρχιτεκτονικής, που προσαρμόζει δυναμικά την τοποθέτηση εργασιών πρόβλεψης, βάσει αξιολόγησης των κινδύνων παραβίασης QoS σε πραγματικό χρόνο. Συνοψίζοντας, οι βασικές συνεισφορές αυτής της εργασίας είναι οι εξής:

- Διεξάγουμε εκτενή **χαρακτηρισμό και ανάλυση** για συντονισμό και ανάπτυξη, βάσει απόδοσης και αρχιτεκτονικής, διακριτών μηχανών πρόβλεψης ML και μοντέλων σε όλο το συνεχές Edge-Cloud.
- **Παρουσιάζουμε το SynergAI, ένα καινοτόμο πλαίσιο χρονοπρογραμματισμού Edge-to-Cloud** για την επίλυση του προβλήματος ελαχιστοποίησης παραβιάσεων QoS. Ενσωματώνουμε έναν συνδυασμό offline και online μηχανισμών στην προτεινόμενη λύση μας, η οποία αξιοποιεί τη διαδικασία χαρακτηρισμού και ανάλυσης, για να εκτελέσει δυναμικό χρονοπρογραμματισμό εργασιών βάσει αξιολογήσεων των κινδύνων παραβίασης QoS σε πραγματικό χρόνο.
- **Ενσωματώνουμε και αξιολογούμε τη λύση μας με το πλαίσιο Kubernetes**, αποδεικνύοντας ότι η αρχιτεκτονικά καθοδηγούμενη εξυπηρέτηση πρόβλεψης του SynergAI καθιστά δυνατές, βελτιστοποιημένες και με επίγνωση αρχιτεκτονικής, αναπτύξεις σε αναδυόμενες πλατφόρμες υλικού, επιτυγχάνοντας μέση μείωση 2.4× στις παραβιάσεις QoS σε σύγκριση με μια λύση state-of-the-art.

Το υπόλοιπο αυτής της διατριβής οργανώνεται ως εξής. Στην Ενότητα 2 παρέχουμε εκτενή χαρακτηρισμό και ανάλυση διακριτών μηχανών πρόβλεψης ML και κόμβων, ενώ στην Ενότητα 3 παρουσιάζουμε την αρχιτεκτονική του SynergAI. Στην Ενότητα 4 παρουσιάζεται και συζητείται η πειραματική αξιολόγηση. Τέλος, η Ενότητα 5 ολοκληρώνει αυτή την έρευνα και προτείνει μελλοντικές ερευνητικές κατευθύνσεις.

2 Χαρακτηρισμός & Ανάλυση

2.1 Πεδίο Δοκιμών Εξυπηρέτησης Προβλέψεων

Υποδομή Υλικού & Λογισμικού: Διεξάγουμε τα πειράματά μας σε έναν υψηλών προδιαγραφών διπλό-επεξεργαστή Intel® Xeon® Gold 5218R (@2.1 GHz) server, εξοπλισμένο με 128 GB μνήμης DRAM. Πάνω σε αυτή τη διάταξη, διαμορφώνουμε δύο εικονικές μηχανές που λειτουργούν ως κύριος κόμβος (4 vCPUs, 8 GB RAM) και κόμβος εργασίας (16 vCPUs, 16 GB RAM) του συστήματός μας, αντίστοιχα, χρησιμοποιώντας τον υπερεπόπτη KVM. Επιπλέον, ενσωματώνουμε

δύο ακόμη κόμβους εργασίας στο Edge, οι οποίοι περιλαμβάνουν ένα Nvidia Jetson AGX (8 CPUs, 32 GB RAM) και ένα Nvidia Jetson Xavier NX (6 CPUs, 8 GB RAM). Για την ανάπτυξη και την ενορχήστρωση του συστήματος, αξιοποιούμε το Kubernetes [17] (v1.28.10) σε συνδυασμό με το Containerd (v1.7.2) ως περιβάλλον εκτέλεσης container.

Φόρτοι Εργασίας των Μηχανών Πρόβλεψης: Για τους σκοπούς αυτής της διατριβής, χρησιμοποιούμε τις εργασίες ανίχνευσης αντικειμένων και ταξινόμησης εικόνων από τη σουίτα benchmark MLPerf Inference [18]. Ο Πίνακας 1.1 παρέχει λεπτομέρειες σχετικά με τις μηχανές πρόβλεψης MLPerf που χρησιμοποιήθηκαν, συμπεριλαμβανομένης της αναπαράστασης, της παραλλαγής του μοντέλου, του συνόλου δεδομένων επικύρωσης και της ακρίβειας του μοντέλου. Κάθε στιγμιότυπο του container της μηχανής πρόβλεψης αποτελείται από δύο στοιχεία: (i) τη Μηχανή Πρόβλεψης και (ii) τη Γεννήτρια Φόρτου. Η Μηχανή Πρόβλεψης επεξεργάζεται ένα προ-εκπαιδευμένο μοντέλο DNN (π.χ., ResNet) χρησιμοποιώντας ένα καθορισμένο πλαίσιο backend (π.χ., TensorFlow) για την εκτέλεση της καθορισμένης εργασίας. Εν τω μεταξύ, η Γεννήτρια Φόρτου προσομοιώνει τη δραστηριότητα για τη Μηχανή Πρόβλεψης και παρακολουθεί την απόδοσή της. Παίρνει ως είσοδο το σύνολο δεδομένων επικύρωσης (π.χ., ImageNet), το σενάριο και τον συνολικό αριθμό ερωτημάτων προς εκτέλεση. Στα πειράματά μας, χρησιμοποιήσαμε το σενάριο Single Stream, όπου η Γεννήτρια Φόρτου στέλνει ένα δείγμα ανά ερώτημα και περιμένει να ολοκληρωθεί η εκτέλεση πριν στείλει το επόμενο. Καθ' όλη τη διάρκεια του χαρακτηρισμού, διατηρούμε έναν σταθερό αριθμό ερωτημάτων στο MLPerf σε όλες τις μηχανές πρόβλεψης για να αξιολογήσουμε την απόδοσή τους υπό τον ίδιο φόρτο εργασίας και να αποκαλύψουμε τα εσωτερικά χαρακτηριστικά τους. Αυτή η τιμή είναι η προεπιλεγμένη που χρησιμοποιείται στο σενάριο Single Stream.

Ανάλυση των Καταστάσεων Λειτουργίας στις Πλακέτες AGX και NX: Οι πλακέτες Nvidia AGX και NX παρέχουν διάφορες καταστάσεις λειτουργίας για τη βελτιστοποίηση της επίδοσης, της ενεργειακής απόδοσης και της θερμικής διαχείρισης. Αυτές οι καταστάσεις επιτρέπουν στο σύστημα να εναλλάσσεται μεταξύ μέγιστης απόδοσης για απαιτητικούς φόρτους εργασίας και χαμηλότερης κατανάλωσης ενέργειας για βελτιωμένη αποδοτικότητα σε περιβάλλοντα με περιορισμένη ενέργεια. Ο Πίνακας 1.2 παρουσιάζει μια λεπτομερή επισκόπηση των καταστάσεων που αναλύθηκαν στην έρευνά μας, όπως προκύπτουν από τις προδιαγραφές του κατασκευαστή. Είναι σημαντικό να σημειωθεί ότι τόσο οι πλακέτες AGX όσο και NX προσφέρουν πρόσθετες καταστάσεις, αλλά η ενεργοποίησή τους απαιτεί επανεκκίνηση. Δεδομένου ότι αυτή η εργασία επικεντρώνεται στη λήψη αποφάσεων κατά τη διάρκεια εκτέλεσης, αποφεύγουμε τη χρήση αυτών των καταστάσεων καθώς εισάγουν επιβαρύνσεις που μπορούν να επηρεάσουν το σύστημα χρονοδρομολόγησης που αναπτύσσουμε. Εστιάζοντας

στο Nvidia Jetson AGX, παρατηρούμε έξι διαθέσιμες καταστάσεις λειτουργίας. Η συχνότητα της CPU κυμαίνεται από 1200 MHz έως 2266 MHz, με τον αριθμό των ενεργών CPUs να ποικίλλει από 2 έως 8. Επιπλέον, ο προϋπολογισμός ισχύος κυμαίνεται από 15 Watt έως MAXN, υποδεικνύοντας ότι δεν υπάρχει σταθερό ανώτατο όριο στην κατανάλωση ενέργειας. Αυτό επιτρέπει στο σύστημα να λειτουργεί σε μέγιστη απόδοση ενώ προσαρμόζει δυναμικά την κατανάλωση ενέργειας σύμφωνα με τις απαιτήσεις του φόρτου εργασίας και τους θερμικούς περιορισμούς. Από την άλλη πλευρά, το Nvidia Jetson Xavier NX προσφέρει εννέα διαθέσιμες καταστάσεις λειτουργίας. Η συχνότητα της CPU εδώ κυμαίνεται από 1200 MHz έως 1900 MHz, με 2 έως 6 ενεργές CPUs, και ο προϋπολογισμός ισχύος κυμαίνεται από 10 Watt έως 20 Watt.

Table 1.1: Υποστηριζόμενες Μηχανές Πρόβλεψης MLPerf, Παραλλαγή Μοντέλου, Σύνολο Δεδομένων και Αντίστοιχη Ακρίβεια

Εργασία	Αναπαράσταση	Παραλλαγή Μοντέλου	Σύνολο Δεδομένων	Ακρίβεια
Ταξινόμηση Εικόνων	Tensorflow (TF) [19]	ResNet50	ImageNet	76.456%
		MobileNet	ImageNet	71.676%
		MobileNet Quantized (Q)	ImageNet	70.694%
	TFLite (TFL) [20]	MobileNet	ImageNet	71.676%
		MobileNet Quantized	ImageNet	70.762%
	ONNX Runtime [21]	ResNet50 `opset-8` (OP8)	ImageNet	76.456%
		ResNet50 `opset-11` (OP11)	ImageNet	76.456%
		MobileNet `opset-8`	ImageNet	71.676%
		MobileNet `opset-11`	ImageNet	71.676%
Ανίχνευση Αντικειμένων	TensorFlow [19]	SSDMobileNet	Coco 300	mAP 0.234
	ONNX Runtime [21]	SSDMobileNet `opset-8`	Coco 300	mAP 0.23
		SSDMobileNet `opset-11`	Coco 300	mAP 0.23

2.2 Χαρακτηρισμός Εξυπηρέτησης Προβλέψεων

Για να αποκτήσουμε βαθύτερη κατανόηση των χαρακτηριστικών εκτέλεσης των διαφόρων μηχανών πρόβλεψης που αναφέρονται στον Πίνακα 1.1, αξιολογούμε την επίδραση διαφόρων παραγόντων στην απόδοσή τους. Συγκεκριμένα, αναλύουμε τις μηχανές για να αξιολογήσουμε πώς i) οι εργάτες βασισμένοι σε x86 και ARM, ii) η κάθετη κλιμάκωση πόρων και iii) οι καταστάσεις λειτουργίας στους εργάτες που βασίζονται σε ARM επηρεάζουν την απόδοσή τους. Στο τέλος αυτής της ανάλυσης, επισημαίνουμε τα βασικά συμπεράσματα από τη μελέτη μας για να προσδιορίσουμε τις βέλτιστες καταστάσεις για υλοποίηση στο σύστημα χρονοδρομολόγησής μας.

2.2.1 Χαρακτηρισμός με Επίγνωση της Απόδοσης

Σε αυτή την ενότητα, μελετάμε τις μηχανές πρόβλεψης που αναφέρονται στον Πίνακα 1.1 για να αναλύσουμε την απόδοσή τους σε όλους τους εργάτες στο σύστημά μας. Το Σχήμα 1.1 παρουσιάζει τον χαρακτηρισμό των αξιολογούμενων μηχανών πρόβλεψης σε όλους τους διαθέσιμους εργάτες. Απεικονίζει (α) το

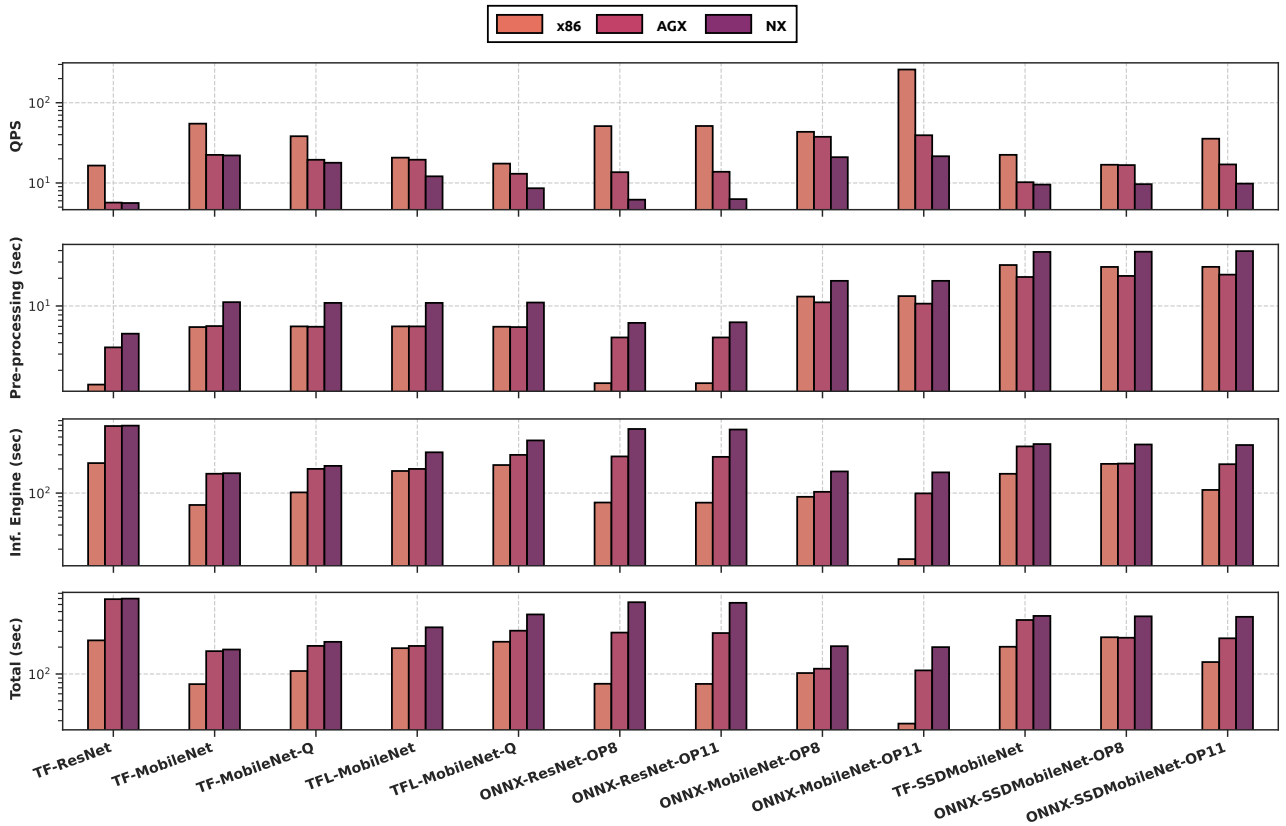


Figure 1.1: Χαρακτηρισμός των Αξιολογούμενων Μηχανών Πρόβλεψης σε Όλους τους Διαθέσιμους Κόμβους: QPS (Πάνω), χρόνος προ-επεξεργασίας (Πάνω-μέση), χρόνος μηχανής πρόβλεψης (Κάτω-μέση) και συνολικός χρόνος εκτέλεσης (Κάτω) σε λογαριθμική κλίμακα.

QPS (Πάνω), (β) τον χρόνο προ-επεξεργασίας, ο οποίος περιλαμβάνει την αρχικοποίηση του backend, τη φόρτωση του μοντέλου DNN και την προετοιμασία των δεδομένων επικύρωσης (Πάνω-μέση), (γ) τον χρόνο εκτέλεσης της μηχανής πρόβλεψης, ο οποίος αναφέρεται στον πραγματικό υπολογισμό χρόνου (Κάτω-μέση), και (δ) τον συνολικό χρόνο εκτέλεσης, όπως καταγράφεται από τη Γεννήτρια Φόρτου για κάθε Μηχανή Πρόβλεψης MLPerf, ο οποίος αντιπροσωπεύει τον συνολικό χρόνο προ-επεξεργασίας και εκτέλεσης της μηχανής πρόβλεψης (Κάτω). Όλες οι μετρήσεις απεικονίζονται σε λογαριθμική κλίμακα. Για τον εργάτη x86, οι τιμές QPS κυμαίνονται από 16.5 για το TensorFlow ResNet έως 259 για το ONNX Runtime MobileNet με `opset-11`, με μέσο όρο QPS 52,3. Για τον συνολικό χρόνο εκτέλεσης, παρατηρούμε μια κατανομή που κυμαίνεται από ελάχιστο 27,8 δευτερόλεπτα για το ONNX Runtime MobileNet με `opset-11` έως μέγιστο 4,3 λεπτά για το ONNX Runtime SSDMobileNet με `opset-8`. Ο μέσος χρόνος εκτέλεσης είναι 2,4 λεπτά. Μια ενδιαφέρουσα παρατήρηση είναι ότι, παρόλο που όλες οι μηχανές πρόβλεψης επεξεργάζονται τον ίδιο αριθμό ερωτημάτων, η μηχανή ONNX ResNet με το χαμηλότερο QPS δεν είναι αυτή με τον μεγαλύτερο χρόνο εκτέλεσης. Αντίθετα, ο συντομότερος χρόνος εκτέλεσης καταγράφεται για το ONNX Runtime MobileNet με `opset-11`, το οποίο έχει QPS 259. Αυτή η συμπεριφορά αποδίδεται στον χρόνο προ-επεξεργασίας, ο οποίος περιλαμβάνει την

επιβάρυνση της αρχικοποίησης του επιλεγμένου backend, την ρύθμιση του προ-εκπαιδευμένου μοντέλου DNN σε αυτό το backend και τη φόρτωση του συνόλου δεδομένων επικύρωσης MLPerf. Πράγματι, το TensorFlow ResNet ολοκληρώνει αυτή τη διαδικασία σε μόλις 1,4 δευτερόλεπτα, ενώ το ONNX Runtime MobileNet με `opset-11` απαιτεί 19× περισσότερο χρόνο για προ-επεξεργασία. Αυτό επηρεάζει σημαντικά την απόδοση, τελικά αντισταθμίζοντας το πλεονέκτημα του υψηλότερου QPS.

Για τον εργάτη AGX, το QPS κυμαίνεται από 5,7 για το TensorFlow ResNet έως 39,3 για το ONNX Runtime MobileNet με `opset-11`, με μέσο όρο QPS 19. Ο συνολικός χρόνος εκτέλεσης εκτείνεται από ελάχιστο 1,8 λεπτά για το ONNX Runtime MobileNet με `opset-11` έως μέγιστο 11,5 λεπτά για το TensorFlow ResNet, με μέσο όρο 4,6 λεπτά. Ομοίως, για τον εργάτη NX, το QPS κυμαίνεται μεταξύ 5,6 για το TensorFlow ResNet και 22 για το TensorFlow MobileNet, με μέσο όρο 12,5. Ο συνολικός χρόνος εκτέλεσης κυμαίνεται από ελάχιστο 3,1 λεπτά για το TensorFlow MobileNet έως μέγιστο 11,6 λεπτά για το TensorFlow ResNet, με μέσο όρο 6,8 λεπτά. Για τους εργάτες AGX και NX, η μηχανή πρόβλεψης με το υψηλότερο QPS οδηγεί στον συντομότερο χρόνο εκτέλεσης, ενώ αυτή με το χαμηλότερο QPS έχει τον μεγαλύτερο χρόνο εκτέλεσης, σε αντίθεση με τον εργάτη x86. Αυτό συμβαίνει επειδή ο χρόνος προ-επεξεργασίας αυξάνεται μόνο ελαφρώς, κατά 10% στο AGX και $1,76\times$ στο NX σε σύγκριση με το x86. Οι εργασίες προ-επεξεργασίας όπως η αλλαγή μεγέθους εικόνας και η κανονικοποίηση βασίζονται περισσότερο στο εύρος ζώνης I/O και μνήμης παρά στην ισχύ της CPU, προκαλώντας ελάχιστη επιβράδυνση σε λιγότερο ισχυρό υλικό. Αντίθετα, η πρόβλεψη είναι πολύ πιο απαιτητική υπολογιστικά, καθιστώντας την αύξηση του χρόνου προ-επεξεργασίας αμελητέα σε σύγκριση με την πολύ μεγαλύτερη αύξηση του χρόνου εκτέλεσης.

⋆ Q1: *Πώς διαφέρει η απόδοση των μηχανών πρόβλεψης όταν εκτελούνται σε διαφορετικούς εργάτες;*

Ο εργάτης x86 ξεπερνά τους AGX και NX, επιτυγχάνοντας $2,8\times$ και $4,2\times$ υψηλότερο QPS, αντίστοιχα, ενώ είναι επίσης $2\times$ και $2,8\times$ ταχύτερος σε χρόνο εκτέλεσης, αντίστοιχα. Αυτό είναι αναμενόμενο, καθώς το x86 προσφέρει την πιο ισχυρή CPU και τη μεγαλύτερη διαθέσιμη RAM, ακολουθούμενο από το AGX και στη συνέχεια το NX. Το TensorFlow ResNet παρουσιάζει σταθερά τη χαμηλότερη απόδοση σε όλες τις συσκευές λόγω της υψηλής υπολογιστικής πολυπλοκότητάς του [22], η οποία γίνεται ακόμη πιο εμφανής σε πλατφόρμες με περιορισμένους πόρους όπως τα AGX και NX. Το ONNX MobileNet `opset-11` είναι η ταχύτερη μηχανή πρόβλεψης τόσο στο x86 όσο και στο AGX, επωφελούμενο από τις βελτιστοποιήσεις του ONNX Runtime για αποδοτική παράλληλη εκτέλεση και πρόβλεψη χαμηλής καθυστέρησης. Ωστόσο, στο NX, το TensorFlow MobileNet

επιτυγχάνει την καλύτερη απόδοση. Αυτή η ανάλυση υπογραμμίζει τη σημασία της ανάπτυξης προβλέψεων με επίγνωση της αρχιτεκτονικής, καθώς η απόδοση μπορεί να διαφέρει σημαντικά σε διαφορετικές πλατφόρμες υλικού, ακόμη και εντός της ίδιας οικογένειας συσκευών, λόγω της αλληλεπίδρασης μεταξύ της αρχιτεκτονικής του συστήματος, των βελτιστοποιήσεων του πλαισίου και των χαρακτηριστικών του μοντέλου πρόβλεψης.

⋆ ***Βασικό Συμπέρασμα 1:*** *Η βέλτιστη επιλογή μηχανής πρόβλεψης και μοντέλου διαφέρει σημαντικά μεταξύ διαφορετικών αρχιτεκτονικών υλικού. Η αποδοτικότητα των προβλέψεων καθορίζεται από τη μηχανή, τα μοντέλα και τα εσωτερικά χαρακτηριστικά της αρχιτεκτονικής.*

2.2.2 Χαρακτηρισμός Συντονισμού Βασισμένου στην Αρχιτεκτονική

Σε αυτή την ενότητα, εξετάζουμε την επίδραση διαφόρων βελτιστοποιήσεων στην απόδοση των αναλυόμενων μηχανών πρόβλεψης. Η προσέγγισή μας χωρίζεται σε δύο μέρη: α) για τον εργάτη x86, αξιολογούμε πώς ο αριθμός των νημάτων επηρεάζει την απόδοση, και β) για τους εργάτες AGX και NX, ερευνούμε πώς οι διαφορετικές καταστάσεις λειτουργίας επηρεάζουν την απόδοση.

⋆ Q2: *Πώς επηρεάζει η κάθετη κλιμάκωση (δηλαδή, #Νήματα) την απόδοση στους εργάτες βασισμένους σε x86;*

Σε αυτό το μέρος της ανάλυσής μας, εξετάζουμε τη συμπεριφορά διαφόρων μηχανών πρόβλεψης όταν εκτελούνται σε έναν εργάτη x86 με διαφορετικό αριθμό νημάτων.

Συγκεκριμένα, για τις μηχανές πρόβλεψης που χρησιμοποιούν το ONNX Runtime ως backend, τροποποιούμε την παράμετρο `INTRA_OP_NUM_THREADS`, και για τις μηχανές που έχουν ως backend το TensorFlow, προσαρμόζουμε τη ρύθμιση `INTRA_OP_PARALLELISM_THREADS`, οι οποίες είναι οι πιο σημαντικές παράμετροι, όπως περιγράφεται σε προηγούμενη έρευνα [4]. Τέλος, για το TensorFlow Lite, εξετάζουμε την παράμετρο `NUM_THREADS`. Τα αποτελέσματα της μελέτης μας έδειξαν ότι η αύξηση του αριθμού των διαθέσιμων νημάτων οδηγεί σε υψηλότερο QPS και μειωμένο συνολικό χρόνο εκτέλεσης. Η χρήση 2, 4, 8 και 16 νημάτων αποδίδει μέσες βελτιώσεις QPS κατά $1,6\times$, $2,5\times$, $3,8\times$ και $4,5\times$ αντίστοιχα, σε σύγκριση με την εκτέλεση μονού νήματος. Όσον αφορά τον συνολικό χρόνο εκτέλεσης, τα 2, 4, 8 και 16 νήματα οδηγούν σε μέσες επιταχύνσεις εκτέλεσης κατά $1,6\times$, $2,3\times$, $2,9\times$ και $3\times$ σε σύγκριση με την εκτέλεση μονού νήματος, αντίστοιχα. Αυτό υποδηλώνει ότι οι βελτιώσεις απόδοσης από την αύξηση των νημάτων δεν είναι πάντα γραμμικές ή σημαντικές πέρα από ένα σημείο. Για παράδειγμα, ενώ η μετάβαση από 1 σε 8 νήματα παρέχει μια επιτάχυνση $2,9\times$, η αύξηση σε 16 νήματα αποδίδει μόνο μια οριακή βελτίωση σε $3\times$. Αυτή η μειούμενη απόδοση δείχνει ότι ύστερα από ένα συγκεκριμένο όριο, η προσθήκη περισσότερων νημάτων δεν βελτιώνει αναλογικά

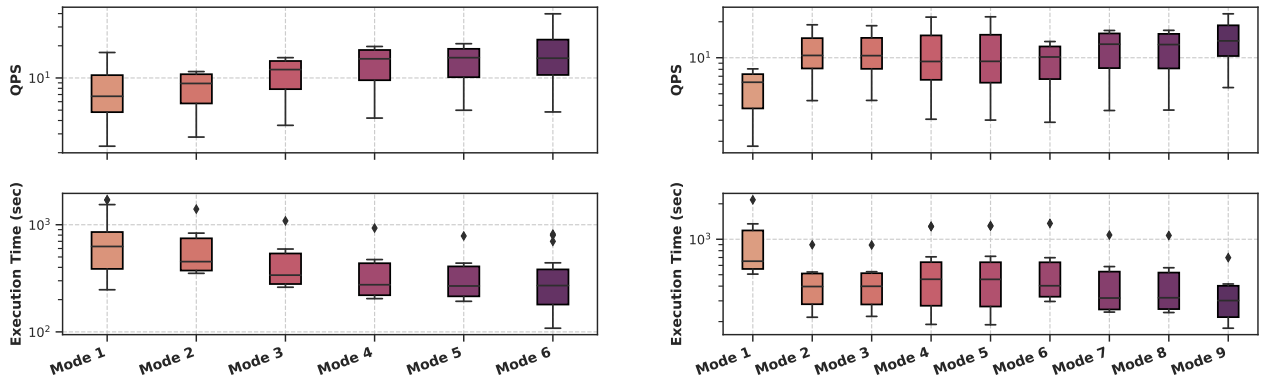
την απόδοση, πιθανότατα λόγω παραγόντων όπως η αυξημένη επιβάρυνση συγχρονισμού, ο ανταγωνισμός για κοινόχρηστους πόρους (π.χ., κρυφή μνήμη ή εύρος ζώνης μνήμης), και ανεπάρκειες στην παράλληλη εκτέλεση. Έτσι, η χρήση όλων των διαθέσιμων νημάτων ενδέχεται να μην είναι πάντα απαραίτητη για την επίτευξη σχεδόν βέλτιστης απόδοσης, ανοίγοντας την πόρτα για περαιτέρω διερεύνηση της συσχέτισης μεταξύ της κλιμάκωσης νημάτων και της αποδοτικότητας των προβλέψεων. Τέλος, ο χρόνος προ-επεξεργασίας δεν επηρεάζεται σημαντικά από την αύξηση των διαθέσιμων νημάτων, με αποτέλεσμα μια μέγιστη μείωση του χρόνου εκτέλεσης κατά 21% στα 16 νήματα σε σύγκριση με την εκτέλεση μονού νήματος. Αυτό είναι αναμενόμενο, καθώς προαναφέραμε ότι η προ-επεξεργασία βασίζεται περισσότερο στο εύρος ζώνης μνήμης και I/O παρά στην CPU.

⋆ ***Βασικό Συμπέρασμα 2:*** *Η αύξηση του αριθμού των νημάτων βελτιώνει την απόδοση πρόβλεψης σε εργάτες βασισμένους σε x86, αλλά με μειωμένη απόδοση πέρα από έναν ορισμένο αριθμό νημάτων, υποδηλώνοντας ότι μπορεί να επιτευχθεί σχεδόν βέλτιστη απόδοση χωρίς τη χρήση όλων των διαθέσιμων νημάτων.*

⋆ Q3: *Παρουσιάζει η απόδοση γραμμική συσχέτιση με την κλιμάκωση νημάτων;*

Όπως παρατηρήθηκε, ο διπλασιασμός του αριθμού των νημάτων δεν οδηγεί σε διπλασιασμό του QPS. Η αύξηση των νημάτων από ένα σημείο και έπειτα αποδίδει φθίνουσες αποδόσεις λόγω ανταγωνισμού νημάτων και επιβαρύνσεων συγχρονισμού. Καθώς περισσότερα νήματα διεκδικούν κοινούς πόρους όπως την κύρια μνήμη και την μνήμη cache, ο ανταγωνισμός αυξάνεται, οδηγώντας σε μειωμένη αποδοτικότητα και περιορισμένη επιτάχυνση. Αυτό το φαινόμενο είναι ακόμη πιο έντονο στον συνολικό χρόνο εκτέλεσης, όπου μετά από μια συγκεκριμένη μείωση στο χρόνο της μηχανής πρόβλεψης, η προεπεξεργασία γίνεται ο κυρίαρχος παράγοντας. Για παράδειγμα, στο ONNX Runtime SSD MobileNet με `opset-8`, από 4 νήματα και μετά, η προεπεξεργασία κυριαρχεί, καθιστώντας τις περαιτέρω αυξήσεις στον αριθμό νημάτων αμελητέες ως προς την επίδραση.

Μια άλλη ενδιαφέρουσα παρατήρηση φαίνεται στο TensorFlow Lite MobileNet Quantized, όπου η απόδοση κλιμακώνεται όπως αναμένεται από 1 έως 8 νήματα, αλλά στα 16 νήματα, τόσο το QPS όσο και ο χρόνος εκτέλεσης παραμένουν σχεδόν πανομοιότυπα με την εκτέλεση με ένα νήμα. Αυτή η συμπεριφορά επηρεάζεται από δύο βασικούς παράγοντες. Πρώτον, το μοντέλο νημάτωσης του TensorFlow Lite είναι σχεδιασμένο για ενσωματωμένες συσκευές, εστιάζοντας στην εκτέλεση χαμηλής καθυστέρησης παρά στον επιθετικό παραλληλισμό. Ως αποτέλεσμα, δεν κατανέμει αποτελεσματικά τους φόρτους εργασίας σε πολλά νήματα, ιδιαίτερα σε επεξεργαστές x86, οι οποίοι δεν είναι ο κύριος στόχος του. Ύστερα από τα 8 νήματα, το μοντέλο νημάτωσης φτάνει σε ένα όριο απόδοσης,



(a) AGX (b) NX

Figure 1.2: Επίδραση των Καταστάσεων Λειτουργίας στο QPS (Πάνω) και χρόνο εκτέλεσης (Κάτω) για τις Μηχανές Πρόβλεψης MLPerf για AGX και NX

προκαλώντας φθίνουσες αποδόσεις στα 16 νήματα. Για να αξιολογήσουμε τη γραμμικότητα, υπολογίζουμε τη συσχέτιση Pearson [23] μεταξύ του χρόνου εκτέλεσης και του αριθμού των νημάτων, με τιμές κοντά στο 1 να υποδεικνύουν ισχυρότερη γραμμική συσχέτιση. Η συσχέτιση μεταξύ του αριθμού των νημάτων και της απόδοσης είναι 0,83 κατά μέσο όρο, υποδεικνύοντας μια ισχυρή αλλά όχι πλήρως γραμμική σχέση. Επιπλέον, η ποσοτικοποίηση παίζει σημαντικό ρόλο στην απόδοση πρόβλεψης σε διαφορετικές πλατφόρμες υλικού. Ενώ οι υπολογισμοί `int8` είναι γενικά ταχύτεροι από τις λειτουργίες κινητής υποδιαστολής λόγω της μειωμένης υπολογιστικής πολυπλοκότητας και των χαμηλότερων απαιτήσεων εύρους ζώνης μνήμης, ο βαθμός των βελτιώσεων απόδοσης ποικίλλει μεταξύ των αρχιτεκτονικών [24, 25]. Δεδομένου ότι οι λειτουργίες `int8` είναι εγγενώς ελαφριές, η υπερβολική νημάτωση μπορεί να εισαγάγει επιβάρυνση συγχρονισμού, ακυρώνοντας πιθανές επιταχύνσεις. Καθώς οι λειτουργίες που βασίζονται σε ακέραιους είναι ελαφριές και εκτελούνται γρήγορα, η επιβάρυνση από τη διαχείριση επιπλέον νημάτων ακυρώνει την πιθανή επιτάχυνση.

⋆ ***Βασικό Συμπέρασμα 3:*** *Η απόδοση δεν κλιμακώνεται πλήρως γραμμικά με τον αριθμό των νημάτων λόγω ανταγωνισμού, επιβαρύνσεων συγχρονισμού και περιορισμών ειδικών για τον φόρτο εργασίας, οδηγώντας σε φθίνουσες αποδόσεις ύστερα από ένα όριο. Έτσι, η υπερκατανάλωση υπολογιστικών πόρων μπορεί να αποφευχθεί.*

⋆ Q4: *Πώς επηρεάζουν οι καταστάσεις λειτουργίας τους εργάτες που βασίζονται σε ARM;*

Η Εικόνα 1.2 απεικονίζει τις κατανομές του QPS και του συνολικού χρόνου εκτέλεσης για τις πλακέτες AGX και NX στις διαθέσιμες καταστάσεις λειτουργίας, όπως περιγράφονται στον Πίνακα 1.2. Εστιάζοντας στον εργάτη AGX, παρατηρούμε ότι η Κατάσταση 6 ξεχωρίζει με το υψηλότερο QPS και, κατά συνέπεια, το χαμηλότερο χρόνο εκτέλεσης. Η κατανομή QPS κυμαίνεται από

4,8 έως 39,8, με μέσο όρο 18,4, ενώ ο χρόνος εκτέλεσης κυμαίνεται από 1,8 έως 13,6 λεπτά, με μέσο όρο 5,1 λεπτά. Συγκεκριμένα, η Κατάσταση 6 παρέχει QPS που είναι $1,3\times$ υψηλότερο από την Κατάσταση 5, $1,4\times$ υψηλότερο από την Κατάσταση 4, $1,8\times$ υψηλότερο από την Κατάσταση 3, $2\times$ υψηλότερο από την Κατάσταση 2, και $2,2\times$ υψηλότερο από την Κατάσταση 1. Αντιθέτως, η πιο αργή κατάσταση, η Κατάσταση 1, παρουσιάζει κατανομή QPS από 4,8 έως 17,3, με μέσο όρο 8,2, με χρόνους εκτέλεσης που κυμαίνονται από 4,1 έως 28,5 λεπτά, με μέσο όρο 11,4 λεπτά. Η Κατάσταση 6 αναμένεται να αποδίδει καλύτερα λόγω της υψηλότερης μέγιστης συχνότητας λειτουργίας της, της αξιοποίησης όλων των διαθέσιμων CPU, και της έλλειψης περιορισμών στον προϋπολογισμό ισχύος. Από την άλλη πλευρά, αν και η Κατάσταση 1 προσφέρει τις μέγιστες online CPUs και υψηλότερο προϋπολογισμό ισχύος, η χαμηλότερη συχνότητά της έχει ως αποτέλεσμα να είναι η πιο αργή επιλογή. Για τον εργάτη NX, η Κατάσταση 9 επιτυγχάνει το υψηλότερο QPS. Η κατανομή QPS κυμαίνεται από 5,6 έως 23,3, με μέσο όρο 14,3, ενώ ο χρόνος εκτέλεσης ποικίλλει μεταξύ 2,9 και 11,6 λεπτών, με μέσο όρο 5,8 λεπτά. Συγκεκριμένα, η Κατάσταση 9 παρέχει QPS που είναι $1,2\times$ υψηλότερο από τις Καταστάσεις 7 και 8, $1,3\times$ υψηλότερο από τις Καταστάσεις 2, 3, 4, και 5, $1,5\times$ υψηλότερο από την Κατάσταση 6, και $2,5\times$ υψηλότερο από την Κατάσταση 1. Η κατάσταση με τη χαμηλότερη απόδοση είναι η Κατάσταση 1, με κατανομή QPS που κυμαίνεται από 1,8 έως 8,1, με μέσο όρο 5,5, και χρόνους εκτέλεσης μεταξύ 8,4 και 36 λεπτών, με μέσο όρο 16,9 λεπτά. Η Κατάσταση 9 επιτυγχάνει υψηλή απόδοση χρησιμοποιώντας τη μεγαλύτερη συχνότητα CPU ενώ χρησιμοποιεί μόνο 4 CPUs και λειτουργεί με τον χαμηλότερο προϋπολογισμό ισχύος. Αντιθέτως, η Κατάσταση 1, η οποία έχει τη χαμηλότερη μέγιστη συχνότητα CPU των 1200 MHz, μαζί με τον ίδιο αριθμό CPU και προϋπολογισμό ισχύος, οδηγεί στην πιο αργή εκτέλεση.

Table 1.2: Διαμορφώσεις Λειτουργίας Ισχύος για τις Πλακέτες Nvidia Jetson AGX Xavier και Xavier NX

Πλακέτα	**Κατάσταση**	**Μέγιστη CPU Συχνότητα (MHz)**	**Online CPUs**	**Διαθέσιμη Ισχύς (W)**
Nvidia Jetson AGX	Κατάσταση 1	1200	8	30
	Κατάσταση 2	1450	6	30
	Κατάσταση 3	1780	4	30
	Κατάσταση 4	2100	2	30
	Κατάσταση 5	2188	4	15
	Κατάσταση 6	2266	8	MAXN
Nvidia Jetson Xavier NX	Κατάσταση 1	1200	4	10
	Κατάσταση 2	1400	4	15
	Κατάσταση 3	1400	4	20
	Κατάσταση 4	1400	6	15
	Κατάσταση 5	1400	6	20
	Κατάσταση 6	1500	2	10
	Κατάσταση 7	1900	2	15
	Κατάσταση 8	1900	2	20
	Κατάσταση 9	1900	4	10

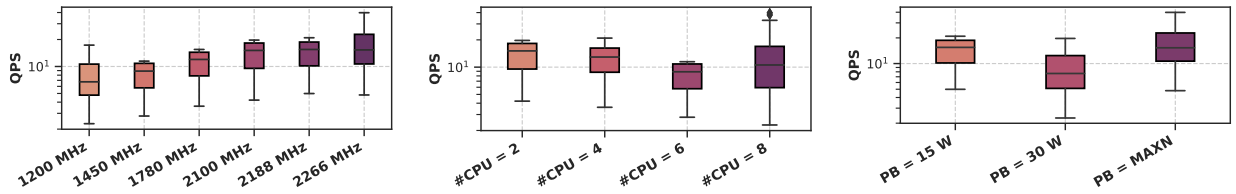


Figure 1.3: Επίδραση της Συχνότητας, #CPUs και Προϋπολογισμού Ισχύος στην Απόδοση για τον Εργάτη AGX

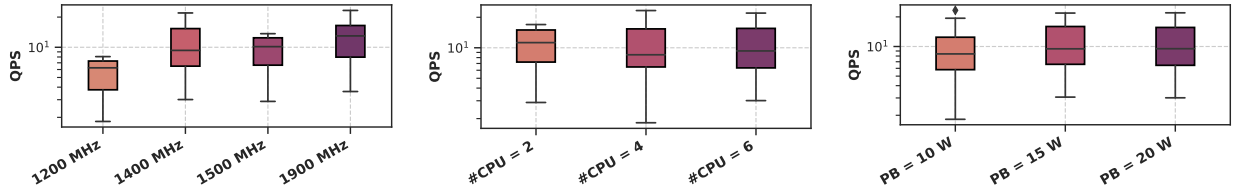


Figure 1.4: Επίδραση της Συχνότητας, #CPUs και Προϋπολογισμού Ισχύος στην Απόδοση για τον Εργάτη NX

⋆ ***Βασικό Συμπέρασμα 4:*** *Οι καταστάσεις λειτουργίας επηρεάζουν σημαντικά την απόδοση σε εργάτες που βασίζονται σε ARM, με τις υψηλότερες συχνότητες CPU να οδηγούν σε καλύτερο QPS και χαμηλότερους χρόνους εκτέλεσης.*

⋆ Q5: *Ποιες παράμετροι (π.χ., #CPUs, συχνότητα) έχουν τη μεγαλύτερη επίδραση στην απόδοση;*

Για να αποκτήσουμε βαθύτερη κατανόηση για τον λόγο που οι προαναφερθείσες καταστάσεις λειτουργίας είναι σημαντικές, αναλύουμε πώς η απόδοση των μηχανών πρόβλεψης επηρεάζεται από τις βασικές πτυχές κάθε κατάστασης λειτουργίας τόσο για τους εργάτες AGX όσο και NX. Οι Εικόνες 1.3 και 1.4 δείχνουν τις κατανομές του QPS σε διαφορετικές μέγιστες συχνότητες λειτουργίας (Αριστερά), τον αριθμό των online CPUs (Κέντρο), και τον διαθέσιμο προϋπολογισμό ισχύος (Δεξιά). Όπως παρατηρείται τόσο για τους εργάτες AGX όσο και NX, η αύξηση της μέγιστης συχνότητας CPU οδηγεί σε υψηλότερη απόδοση. Για τον εργάτη AGX, η αύξηση της συχνότητας CPU από 1200 MHz σε 1450 MHz οδηγεί σε μέση αύξηση του QPS κατά $2,5\%$. Η αύξηση στα 1780 MHz οδηγεί σε βελτίωση κατά $1,35\times$, στα 2100 MHz σε αύξηση κατά $1,7\times$, στα 2188 MHz σε αύξηση κατά $1,8\times$, και στα 2266 MHz σε βελτίωση κατά $2,3\times$ κατά μέσο όρο. Παρόμοια συμπεριφορά παρατηρείται για τον εργάτη NX, όπου η μετάβαση από την ίδια αρχική συχνότητα CPU στην επόμενη διαθέσιμη συχνότητα οδηγεί σε $1,7\times$, $2\times$, και $2,3\times$ υψηλότερο QPS κατά μέσο όρο.

Όσον αφορά τον αριθμό των διαθέσιμων online CPUs, παρατηρούμε μια αντιφατική τάση. Για τον εργάτη AGX, η αύξηση του αριθμού των online CPUs γενικά οδηγεί σε μείωση του QPS κατά μέσο όρο. Συγκεκριμένα, η χρήση 4 online CPUs οδηγεί σε μείωση του QPS κατά 8%, ενώ η ύπαρξη 6 online CPUs

οδηγεί σε $1,7\times$ χαμηλότερο QPS σε σύγκριση με τη χρήση 2 online CPUs. Με 8 online CPUs, το QPS παραμένει σχεδόν το ίδιο όπως με 2 CPUs, δείχνοντας μόνο μια ελαφρά πτώση 1% κατά μέσο όρο. Για τον εργάτη NX, το QPS παραμένει σχετικά σταθερό. Παρατηρούμε μια ελαφρά μείωση 2% κατά μέσο όρο με 4 CPUs, ακολουθούμενη από αύξηση 4% όταν χρησιμοποιούμε 6 CPUs σε σύγκριση με τη διαμόρφωση των 2 CPUs. Για να κατανοήσουμε αυτή τη συμπεριφορά, εστιάζουμε στις καταστάσεις λειτουργίας με 2 CPUs. Στον εργάτη AGX, η διαμόρφωση με 2 CPUs αντιστοιχεί στην Κατάσταση 4, η οποία λειτουργεί σε υψηλή συχνότητα 2100 MHz. Αντιθέτως, η διαμόρφωση με 6 CPUs, όπου παρατηρούμε τη μεγαλύτερη πτώση QPS, περιλαμβάνει μόνο την Κατάσταση 2, η οποία έχει χαμηλότερη συχνότητα 1450 MHz. Ακόμη και με 8 online CPUs, συμπεριλαμβανομένης της Κατάστασης 6 με την καλύτερη απόδοση, η παρουσία της Κατάστασης 1 που λειτουργεί σε χαμηλότερη συχνότητα 1200 MHz προκαλεί μείωση στην κατανομή QPS. Παρόμοιες παρατηρήσεις ισχύουν για τον εργάτη NX. Αυτό υποδηλώνει ότι η συχνότητα λειτουργίας είναι ένας πιο κρίσιμος παράγοντας από τον αριθμό των online CPUs. Εάν η απόδοση είναι ο πρωταρχικός στόχος, είναι συχνά προτιμότερο να επιλέξουμε μια κατάσταση λειτουργίας με λιγότερες CPU αλλά υψηλότερη συχνότητα, παρά να αυξήσουμε τον αριθμό των CPU με κόστος τη μειωμένη συχνότητα.

Όσον αφορά τον διαθέσιμο προϋπολογισμό ισχύος των καταστάσεων λειτουργίας για τον εργάτη AGX, ο προϋπολογισμός ισχύος MAXN ξεχωρίζει με μέσο QPS 18,4. Ο επόμενος καλύτερος προϋπολογισμός ισχύος είναι 15 Watt, ο οποίος οδηγεί σε μείωση κατά $1,3\times$, ακολουθούμενος από 30 Watt, ο οποίος οδηγεί σε μείωση κατά $2\times$ σε σύγκριση με το MAXN. Αν και θα περίμενε κανείς ότι η κατάσταση 30-Watt θα απέδιδε καλύτερα από την κατάσταση 15-Watt, μια βαθύτερη ανάλυση των καταστάσεων λειτουργίας αποκαλύπτει ότι ο προϋπολογισμός ισχύος 15-Watt συνδέεται αποκλειστικά με την Κατάσταση 6, η οποία έχει τη δεύτερη υψηλότερη συχνότητα λειτουργίας. Αντιθέτως, ο προϋπολογισμός ισχύος 30-Watt περιλαμβάνει πολλαπλές καταστάσεις με σημαντικά χαμηλότερες συχνότητες, οδηγώντας σε χαμηλότερο μέσο QPS. Για τον εργάτη NX, τα αποτελέσματα είναι τα αναμενόμενα, με τις χαμηλότερες τιμές QPS να παρατηρούνται στα 10 Watt, με μέσο όρο 9,7 QPS. Στα 15 Watt και 20 Watt, οι τιμές QPS είναι παρόμοιες, με μέσο όρο 11,3. Η ομοιότητα στην απόδοση μεταξύ 15 Watt και 20 Watt αποδίδεται στο ευρύτερο φάσμα συχνοτήτων CPU που είναι διαθέσιμες εντός αυτών των προϋπολογισμών ισχύος, εμποδίζοντας οποιαδήποτε μεμονωμένη κατάσταση να ξεχωρίσει.

⋆ ***Βασικό Συμπέρασμα 5:*** *Η συχνότητα CPU έχει τη μεγαλύτερη επίδραση στην απόδοση, υπερτερώντας του αριθμού των online CPUs, ενώ ο προϋπολογισμός ισχύος επηρεάζει την απόδοση έμμεσα με βάση τη συχνότητα και τις καταστάσεις που επιτρέπει.*

3 Πλαίσιο χρονοπρογραμματισμού του SynergAI

Με βάση τις γνώσεις που προέκυψαν από τον χαρακτηρισμό και την ανάλυση που παρουσιάζονται στην Ενότητα 2.2, σχεδιάζουμε το `SynergAI`. Ο κύριος στόχος του είναι να ικανοποιήσει τις απαιτήσεις QoS των εγκατεστημένων μηχανών πρόβλεψης, μειώνοντας τον αριθμό των παραβιάσεων QoS. Το `SynergAI` απεικονίζεται στην Εικόνα 1.5 και στοχεύει να παρέχει μια συνεργατική λύση Edge-to-Cloud και αποτελείται από δύο διακριτές φάσεις, συγκεκριμένα i) *Ανάλυση Απόδοσης & Χαρακτηρισμός με βάση την Αρχιτεκτονική (Offline)* που περιγράφεται λεπτομερώς στην Ενότητα 3.1 και ii) *Χρονοπρογραμματισμός & Ανάπτυξη με επίγνωση QoS σε Πραγματικό Χρόνο (Online)*, που αναλύεται στην Ενότητα 3.2.

3.1 Offline Φάση: Ανάλυση Απόδοσης & Χαρακτηρισμός με βάση την Αρχιτεκτονική

Η offline φάση στοχεύει στην αξιολόγηση του τρόπου με τον οποίο κάθε μηχανή πρόβλεψης αποδίδει κάτω από διάφορες βελτιστοποιήσεις ειδικές για την αρχιτεκτονική, με γνώμονα τον χαρακτηρισμό και την ανάλυση που παρουσιάζονται στην Ενότητα 2.2. Ως είσοδο παρέχουμε τις μηχανές πρόβλεψης-στόχους που προορίζονται για ανάπτυξη εντός του προτεινόμενου συστήματος εξυπηρέτησης προβλέψεων, καθώς και τους κόμβους-στόχους Edge/Cloud. Πιο συγκεκριμένα, η μηχανή πρόβλεψης αποτελείται από το backend (π.χ., TensorFlow, ONNX Runtime), το προ-εκπαιδευμένο μοντέλο DNN (π.χ., ResNet) και το σύνολο δεδομένων επικύρωσης (π.χ., ImageNet), ενώ οι κόμβοι-στόχοι χαρακτηρίζονται από την υποκείμενη αρχιτεκτονική των εργατών (π.χ., x86, ARM) και τις καταστάσεις λειτουργίας τους. Το αποτέλεσμα της offline φάσης είναι η δημιουργία ενός λεξικού διαμόρφωσης για τις διακριτές μηχανές πρόβλεψης, το οποίο ενσωματώνει τα δεδομένα που θα χρησιμοποιηθούν αργότερα για τον χρονοπρογραμματισμό εργασιών κατά την Online φάση (Ενότητα 3.2).

Αρχικά, η *Γεννήτρια Διαμόρφωσης με επίγνωση Απόδοσης* (1A) εξετάζει τα διαθέσιμα επίπεδα παραλληλισμού για κάθε αρχιτεκτονική διαμόρφωση, βελτιστοποιώντας αντίστοιχα την αποδοτικότητα εκτέλεσης. Αυτή η διαδικασία περιλαμβάνει τη διαμόρφωση του βαθμού κάθετης κλιμάκωσης για κάθε backend εντός της μηχανής πρόβλεψης σε κάθε κόμβο-εργάτη, όπως τονίζεται στην προηγούμενη έρευνα στο [4]. Μέσω της συστηματικής διερεύνησης των επιλογών παραλληλισμού, το σύστημα διασφαλίζει ότι οι φόρτοι εργασίας πρόβλεψης κατανέμονται βέλτιστα σε υπολογιστικούς πόρους, μεγιστοποιώντας τη διεκπεραιωτική ικανότητα και ελαχιστοποιώντας την καθυστέρηση. Εκτός από τη βελτιστοποίηση παραλληλισμού, το `SynergAI` ενσωματώνει την *Γεννήτρια Διαμόρφωσης με επίγνωση Αρχιτεκτονικής* (1B), σχεδιασμένο να βελτιώσει την αποδοτικότητα σε αρχιτεκτονικές όπου είναι εφικτή η ρύθμιση καταστάσεων λειτουργίας. Αυτή η μονάδα

ενσωματώνει τις διακριτές καταστάσεις λειτουργίας που διατίθενται σε συγκεκριμένες πλατφόρμες υλικού, όπως οι πλακέτες AGX και NX, για να αξιολογήσει τον αντίκτυπό τους στην απόδοση εντός διακριτών ορίων ισχύος. Επιλέγοντας δυναμικά την πιο αποδοτική διαμόρφωση κατάστασης λειτουργίας, το SynergAI διασφαλίζει ότι οι φόρτοι εργασίας πρόβλεψης εκτελούνται με βέλτιστη ισορροπία μεταξύ υπολογιστικής ταχύτητας, διαχείρισης νημάτων και αποδοτικότητας, οδηγώντας σε υλοποιήσεις συντονισμένες με την αρχιτεκτονική. Αυτή η διπλή στρατηγική βελτιστοποίησης—που αξιοποιεί τόσο την εξερεύνηση παραλληλισμού όσο και τη ρύθμιση με επίγνωση αρχιτεκτονικής—επιτρέπει στο SynergAI να διαμορφώνει δυναμικά ετερογενείς κόμβους Edge και Cloud, βελτιστοποιώντας τη χρήση πόρων ενώ ικανοποιεί τους περιορισμούς QoS.

Μόλις συγκεντρωθούν όλες οι πιθανές διαμορφώσεις για μια μηχανή πρόβλεψης στους διαθέσιμους εργάτες, πραγματοποιείται μια ενδελεχής *Εξερεύνηση & Ανάλυση Χώρου Σχεδιασμού* 1C. Αυτό το βήμα περιλαμβάνει τη συστηματική αξιολόγηση κάθε διαμόρφωσης για να εντοπιστούν οι πιο αποτελεσματικές και υψηλής απόδοσης ρυθμίσεις για διαφορετικούς εργάτες, λαμβάνοντας υπόψη τις συγκεκριμένες καταστάσεις λειτουργίας. Με την προσεκτική εξερεύνηση αυτών των διαμορφώσεων, αποκτούμε πολύτιμες γνώσεις σχετικά με τις αντισταθμίσεις μεταξύ ταχύτητας και αποδοτικότητας, καταλήγοντας τελικά σε μια καλά ισορροπημένη και βελτιστοποιημένη στρατηγική ανάπτυξης. Κατά τη διάρκεια αυτής της εξερεύνησης, συλλέγουμε λεπτομερείς μετρήσεις απόδοσης, όπως το επιτευχθέν QPS, ο συνολικός χρόνος εκτέλεσης, ο χρόνος προ-επεξεργασίας, ο πραγματικός χρόνος υπολογισμού, τα νήματα και άλλα. Αυτές οι μετρήσεις παρέχουν μια ολοκληρωμένη άποψη της απόδοσης της μηχανής πρόβλεψης υπό διάφορες διαμορφώσεις. Μετά από μια εις βάθος ανάλυση των συλλεχθέντων δεδομένων, προσδιορίζουμε τις *Βέλτιστες Υλοποιήσεις με βάση την Αρχιτεκτονική* 1D για κάθε μηχανή πρόβλεψης, επιλέγοντας τις διαμορφώσεις καλύτερης απόδοσης από όλες τις διαθέσιμες αρχιτεκτονικές. Αυτά τα δεδομένα αποθηκεύονται στη συνέχεια σε μια δομημένη βάση δεδομένων, εξασφαλίζοντας εύκολη πρόσβαση και ανάκτηση όταν απαιτείται. Το τελικό σύνολο δεδομένων σχηματίζει το *Λεξικό Διαμόρφωσης* 1E, το οποίο χρησιμεύει ως αναφορά για τον ενορχηστρωτή SynergAI. Αυτό το λεξικό περιλαμβάνει κρίσιμα στοιχεία για βέλτιστη εξυπηρέτηση προβλέψεων, όπως το μοντέλο, το backend, την κατάσταση λειτουργίας και άλλες παραμέτρους του συστήματος. Λειτουργεί ως βάση για τον χρονοπρογραμματισμό εργασιών στην *Online Φάση* (Ενότητα 3.2), επιτρέποντας στον ενορχηστρωτή να λαμβάνει ευφυείς, συντονισμένες με την αρχιτεκτονική αποφάσεις που μεγιστοποιούν την αποδοτικότητα ολόκληρου του συστήματος Edge-Cloud.

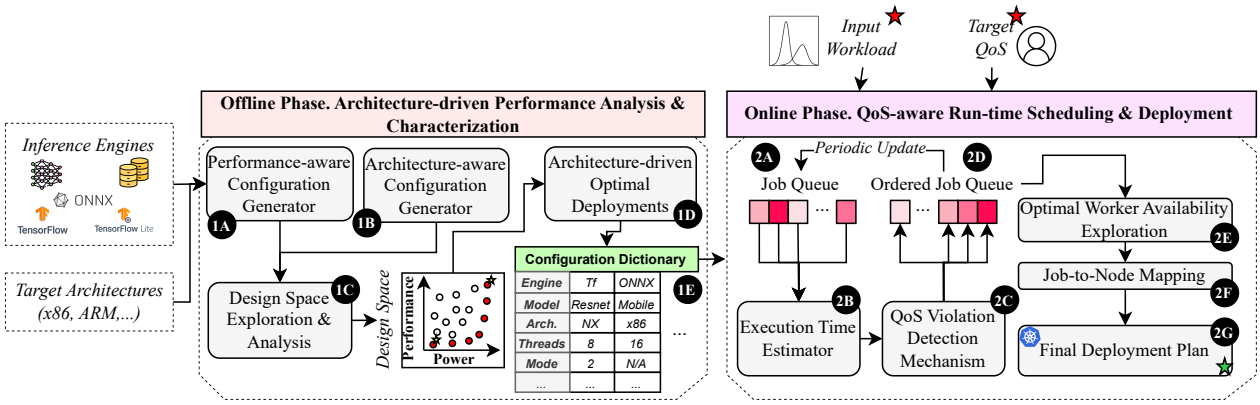


Figure 1.5: Επισκόπηση των Offline και Online Φάσεων του Πλαισίου SynergAI

3.2 Online Φάση: Χρονοπρογραμματισμός & Ανάπτυξη με επίγνωση QoS σε Πραγματικό Χρόνο

Κατά τη διάρκεια της Online Φάσης, το SynergAI επεξεργάζεται συνεχώς εισερχόμενους φόρτους εργασίας πρόβλεψης, διασφαλίζοντας παράλληλα τη συμμόρφωση με τους στόχους QoS. Η πολιτική χρονοπρογραμματισμού κατασκευάζει μια *Ουρά Εργασιών* 2A, όπου κάθε εργασία αντιπροσωπεύει μια μηχανή πρόβλεψης με συγκεκριμένες απαιτήσεις εκτέλεσης, συμπεριλαμβανομένου του συνολικού αριθμού ερωτημάτων που πρέπει να υποβληθούν σε επεξεργασία και του στόχου QoS. Έστω J το σύνολο των εισερχόμενων εργασιών, ορισμένο ως $J = \{j_1, j_2, \ldots, j_N\}$, και W το σύνολο των κόμβων-εργατών, ορισμένο ως $W = \{w_1, w_2, \ldots, w_M\}$. Η διατύπωση του προβλήματος-στόχου μας και η διαδικασία λήψης αποφάσεων παρουσιάζονται στις Εξισώσεις 1.1-1.4, όπου κάθε εξίσωση αξιολογείται για κάθε εργασία $j \in J$. Οι βασικές παράμετροι εξηγούνται σύντομα στον Πίνακα 1.3.

$$T_{\text{Remaining},j} = T_{\text{QoS},j} - T_{\text{Waiting},j}, \quad \forall j \in J \tag{1.1}$$

$$T_{\text{Estimated},j,w} = T_{\text{Pre-processing},j} + \frac{q}{QPS^{c^*_{j,w}}}, \quad \forall j \in J, \forall w \in W \tag{1.2}$$

$$W_{\text{acceptable},j} = \{w \in W \mid T_{\text{Remaining},j} \geq T_{\text{Estimated},j,w}\}, \quad \forall j \in J \tag{1.3}$$

$$w^*_j = \arg \min_{w \in W_{\text{acceptable},j}} T_{\text{Estimated},j,w} \tag{1.4}$$

Table 1.3: Βασικές Παράμετροι Συστήματος και Χρονοπρογραμματιστή

Συμβολισμός	**Σύντομη Περιγραφή**
q	Αριθμός ερωτημάτων προς εκτέλεση για τη δεδομένη πρόβλεψη
J	Σύνολο εργασιών πρόβλεψης στο σύστημα, όπου κάθε εργασία $j \in J$.
W	Σύνολο κόμβων worker στο σύστημα, όπου κάθε worker $w \in W$.
$c^{j,w}$	Βέλτιστη διαμόρφωση για την εργασία j στον worker w, μεγιστοποιώντας το QPS.
w^j	Ο βέλτιστος worker για την εργασία j που ελαχιστοποιεί τον χρόνο εκτέλεσης ενώ ικανοποιεί τους περιορισμούς QoS.
$T\text{QoS}, j$	Χρόνος που καθορίζεται από τον χρήστη για την εκτέλεση της εργασίας j.
$T_{\text{Waiting},j}$	Χρόνος που έχει παρέλθει από τότε που η εργασία j υποβλήθηκε στην ουρά.
$T_{\text{Remaining},j}$	Υπολειπόμενος χρόνος πριν συμβεί παραβίαση QoS για την εργασία j, υπολογίζεται ως $T_{\text{QoS},j} - T_{\text{Waiting},j}$.
$T_{\text{Pre-processing},j}$	Χρόνος προ-επεξεργασίας για την εργασία j, που προκύπτει από την καταγραφή προφίλ.
$T_{\text{Estimated},j,w}$	Εκτιμώμενος χρόνος εκτέλεσης για την εργασία j στον worker w, συμπεριλαμβανομένου του χρόνου προ-επεξεργασίας και του χρόνου εκτέλεσης ανά ερώτημα.
$QPS^{c^{j,w}}$	Ερωτήματα ανά Δευτερόλεπτο (QPS) που επιτυγχάνονται από τη βέλτιστη διαμόρφωση $c^*_{j,w}$ για την εργασία j στον worker w.
$W\text{acceptable}, j$	Σύνολο workers ικανών να ολοκληρώσουν την εργασία j εντός του υπολειπόμενου επιτρεπόμενου χρόνου.

Έστω $T_{\text{QoS},j}$ ο χρόνος που καθορίζεται από τον χρήστη για την εκτέλεση της εργασίας j, και έστω $T_{\text{Waiting},j}$ ο χρόνος που έχει παρέλθει από τότε που η εργασία j υποβλήθηκε στην ουρά. Ορίζουμε τον υπολειπόμενο χρόνο πριν από μια παραβίαση QoS για το j ως $T_{\text{Remaining},j}$, που εκφράζεται στην Εξίσωση 1.1. Καθώς το $T_{\text{Waiting},j}$ αυξάνεται, το $T_{\text{Remaining},j}$ μειώνεται, πλησιάζοντας το μηδέν, υποδεικνύοντας αυξανόμενη επείγουσα ανάγκη για εκτέλεση. Για την αποτελεσματική διαχείριση της ιεράρχησης εργασιών και την αποφυγή παραβιάσεων QoS, το `SynergAI` παρακολουθεί συνεχώς αυτούς τους χρονικούς περιορισμούς. Στη συνέχεια, οι εργασίες στην ουρά προωθούνται στον *Εκτιμητή Χρόνου Εκτέλεσης* 2B, ο οποίος είναι ο βασικός παράγοντας που επιτρέπει τον προσαρμοστικό χρονοπρογραμματισμό, επιτρέποντας στο `SynergAI` να καθορίζει δυναμικά τη σειρά εκτέλεσης εργασιών. Χρησιμοποιώντας το *Λεξικό Διαμόρφωσης* 1D, το `SynergAI` επιλέγει τη βέλτιστη διαμόρφωση για κάθε κόμβο-εργάτη που μεγιστοποιεί το QPS για μια δεδομένη μηχανή πρόβλεψης, που συμβολίζεται ως $c^*_{j,w}$ για κάθε εργασία j και εργάτη w. Δεδομένου ενός αιτήματος για εκτέλεση q ερωτημάτων και του καταγεγραμμένου χρόνου προ-επεξεργασίας για την εργασία j, που συμβολίζεται ως $T_{\text{Pre-processing},j}$, ο εκτιμώμενος χρόνος εκτέλεσης $T_{\text{Estimated},j,w}$ διατυπώνεται στην Εξίσωση 1.2, όπου το $QPS^{c^*_{j,w}}$ αντιπροσωπεύει το QPS που επιτυγχάνεται από τη διαμόρφωση $c^*_{j,w}$. Αυτή η πληροφορία ενσωματώνεται στον *Μηχανισμό Ανίχνευσης Παραβιάσεων QoS* 2C, του οποίου στόχος είναι να εντοπίζει εργασίες που κινδυνεύουν να υπερβούν τους περιορισμούς QoS τους, να καθορίζει τον βέλτιστο εργάτη για ανάπτυξη και να δίνει προτεραιότητα σε εργασίες με υψηλότερη πιθανότητα παραβίασης. Το σύστημα αξιολογεί το πόσο επείγει κάθε εργασία υπολογίζοντας τη διαφορά μεταξύ $T_{\text{Remaining},j}$ και $T_{\text{Estimated},j,w}$ για κάθε εργάτη w. Καθώς η διαφορά πλησιάζει το μηδέν, η προτεραιότητα αυξάνεται, ενώ

μια αρνητική τιμή υποδεικνύει μια αναπόφευκτη παραβίαση QoS.

Η τελική επιλογή κόμβου-εργάτη καθορίζεται μέσω ενός συνόλου αποδεκτών εργατών, οι οποίοι μπορούν να εγγυηθούν τη συμμόρφωση με το QoS δεδομένου του υπολειπόμενου χρόνου $T_{\text{Remaining},j}$, όπως ορίζεται στην Εξίσωση 1.3. Αυτοί οι κόμβοι-εργάτες στη συνέχεια ταξινομούνται κατά αύξουσα σειρά με βάση τον εκτιμώμενο χρόνο εκτέλεσής τους. Ο βέλτιστος εργάτης w_j^* για τη δεδομένη εργασία είναι εκείνος που ελαχιστοποιεί τον εκτιμώμενο χρόνο εκτέλεσης $T_{\text{Estimated},j,w}$, όπως διατυπώνεται στην Εξίσωση 1.4. Αυτό εγγυάται ότι οι εργασίες ανατίθενται στους ταχύτερους διαθέσιμους κόμβους-εργάτες, μειώνοντας τον χρόνο εκτέλεσης ενώ πληρούν τις απαιτήσεις QoS. Επιπλέον, διατηρώντας μια ταξινομημένη λίστα αποδεκτών κόμβων για κάθε εργασία, αποτρέπουμε αυξημένους χρόνους αναμονής ελέγχοντας τον επόμενο καλύτερο διαθέσιμο κόμβο όταν η πρώτη επιλογή είναι κατειλημμένη. Μόλις υπολογιστούν οι βαθμοί προτεραιότητας για όλες τις εργασίες και προσδιοριστούν οι βέλτιστες αναθέσεις, η ουρά ταξινομείται κατά φθίνουσα σειρά με βάση το πόσο επείγουσα είναι η κάθε εργασία, διασφαλίζοντας ότι οι εργασίες με τον υψηλότερο κίνδυνο παραβιάσεων QoS έχουν προτεραιότητα, συνθέτοντας την *Ταξινομημένη Ουρά Εργασιών* 2D. Εάν κανένας εργάτης δεν μπορεί να ολοκληρώσει μια εργασία εντός του απαιτούμενου χρόνου εκτέλεσής της (δηλαδή, συμβαίνει μια παραβίαση QoS), η εργασία χάνει την προτεραιότητά της και μετακινείται στο τέλος της ουράς, επιτρέποντας πρώτα την επεξεργασία εργασιών υψηλότερης προτεραιότητας με ενεργούς περιορισμούς QoS. Για τη διασφάλιση συνεχούς βελτιστοποίησης, ένας μηχανισμός περιοδικής ενημέρωσης επαναξιολογεί όλες τις εργασίες στην ουρά, αντιστοιχίζοντάς τες στον πιο κατάλληλο εργάτη με βάση τους ενημερωμένους χρόνους αναμονής και επαναταξινομώντας δυναμικά την ουρά όπως απαιτείται. Οι εργασίες αφαιρούνται διαδοχικά από την ουρά, καθεμία συσχετισμένη με μια λίστα ζευγών $(w, c_{j,w}^*)$, όπου κάθε ζεύγος αποτελείται από έναν κόμβο-εργάτη και τη βέλτιστη διαμόρφωσή του για τη δεδομένη μηχανή πρόβλεψης. Τέλος, η *Εξερεύνηση Διαθεσιμότητας Βέλτιστου Κόμβου-Εργάτη* 2E εξετάζει το ταξινομημένο σύνολο $W_{\text{acceptable},j}$, ελέγχοντας τη διαθεσιμότητα κάθε εργάτη μέχρι να βρει τον πρώτο διαθέσιμο: $(w_j,\ c_{j,w_j^*}^*)$, ξεκινώντας από τον βέλτιστο εργάτη w_j^* και στη συνέχεια εκτελεί την τελική *Αντιστοίχιση Εργασίας-σε-Κόμβο* 2F. Μόλις η εργασία ανατεθεί στον αντίστοιχο κόμβο, το *Τελικό Σχέδιο Ανάπτυξης* 2G προκύπτει ανάλογα και αναπτύσσεται μέσω του Kubernetes. Αυτή η επαναληπτική διαδικασία συνεχίζεται μέχρι όλες οι εργασίες να χρονοπρογραμματιστούν και να εκτελεστούν επιτυχώς εντός του συμπλέγματος Edge/Cloud.

4 Πειραματική Αξιολόγηση

Σε αυτή την ενότητα, παρουσιάζουμε την πειραματική αξιολόγηση του SynergAI. Αρχικά, εξηγούμε την πειραματική διάταξη και τους μηχανισμούς που χρησιμοποιούνται για σύγκριση. Τέλος, παρουσιάζουμε και αναλύουμε τα αποτελέσματα των πειραμάτων μας και τον αντίκτυπο του SynergAI.

4.1 Επισκόπηση Πειραμάτων και Περιγραφή Σημείων Αναφοράς

Αξιολογούμε τον χρονοπρογραμματιστή SynergAI μέσω πολλαπλών πειραμάτων για να εκτιμήσουμε την αποδοτικότητά του. Κάθε πείραμα αποτελείται από ένα σύνολο 24 μηχανών πρόβλεψης προς εξυπηρέτηση. Κάθε μηχανή πρόβλεψης καθορίζει έναν απαιτούμενο αριθμό ερωτημάτων προς εκτέλεση μαζί με τους περιορισμούς QoS, που σημαίνει ότι αναμένει να ολοκληρώσει αυτά τα ερωτήματα εντός καθορισμένου χρονικού ορίου. Για τον καθορισμό των απαιτήσεων, συγκεντρώσαμε τους χρόνους εκτέλεσης όλων των διαμορφώσεων και κόμβων για κάθε μηχανή πρόβλεψης, λαμβάνοντας υπόψη τα διαφορετικά φορτία των 3900, 1950, 975 και 488 ερωτημάτων ανά πειραματική εκτέλεση. Από αυτή την κατανομή, εξαγάγαμε τις διάμεσες τιμές για να ορίσουμε ένα σύνολο απαιτήσεων χαμηλής έντασης (DL) και τις τιμές του 25%-ου εκατοστημορίου για να αντιπροσωπεύσουμε ένα φόρτο εργασίας υψηλής έντασης απαιτήσεων (DH). Όσον αφορά την άφιξη των αιτημάτων πρόβλεψης στον χρονοπρογραμματιστή του SynergAI, ακολουθούμε μια κατανομή Poisson, όπως γίνεται σε προηγούμενες έρευνες [1], καθώς μοντελοποιεί αποτελεσματικά τις αφίξεις αιτημάτων καταγράφοντας ανεξάρτητα γεγονότα, ακολουθώντας εκθετικό χρόνο μεταξύ αφίξεων, αντικατοπτρίζοντας την μεταβλητότητα του πραγματικού κόσμου και προσφέροντας μαθηματική απλότητα για ανάλυση. Η παράμετρος λ της κατανομής προέρχεται από τα δεδομένα που συλλέχθηκαν κατά τη διάρκεια του χαρακτηρισμού. Συγκεκριμένα, συγκεντρώνουμε τους χρόνους εκτέλεσης για όλες τις μηχανές πρόβλεψης σε όλες τις διαμορφώσεις και τους κόμβους, εξάγοντας τις διάμεσες τιμές και τις τιμές του 25%-ου εκατοστημορίου από την κατανομή για να ορίσουμε μια χαμηλή συχνότητα αιτημάτων (FL) και μια υψηλή συχνότητα αιτημάτων (FH). Συνολικά, δημιουργούμε τέσσερα διακριτά πειράματα: DL−FL, DL−FH, DH−FL, και DH−FH, καθένα με αυξανόμενη δυσκολία για την αξιολόγηση του συστήματος χρονοπρογραμματισμού μας.

Συγκρίνουμε τον χρονοπρογραμματιστή SynergAI με διάφορες άλλες μεθόδους χρονοπρογραμματισμού, από τυπικές προσεγγίσεις έως state-of-the-art συστήματα χρονοπρογραμματισμού. Εκτός από την προσέγγιση χρονοπρογραμματισμού μας, υλοποιούμε πέντε επιπλέον συστήματα χρονοπρογραμματισμού: i) *Round Robin* (RR), το οποίο κατανέμει μηχανές πρόβλεψης σε εργάτες σε κυκλική ακολουθία, ανεξάρτητα από τις δυνατότητες των εργατών ή τις απαιτήσεις εργασίας, εξασφαλίζοντας δίκαιη κατανομή του φόρτου εργασίας σε όλο το σύστημα· ii)

Strict Round Robin (SRR), μια παραλλαγή του RR, όπου κάθε εργασία ανατίθεται αυστηρά στον επόμενο κόμβο-εργάτη και περιμένει τη διαθεσιμότητά του, στοχεύοντας σε τέλεια κατανομή αλλά διακινδυνεύοντας αυξημένους χρόνους αναμονής και παραβιάσεις απαιτήσεων · iii) *Least Recently Used* (LRU), το οποίο αναθέτει εργασίες στον κόμβο που ήταν αδρανής για το μεγαλύτερο χρονικό διάστημα για την πρόληψη της εξάντλησης και την προώθηση ομοιόμορφης κατανομής φόρτου εργασίας · iv) *Most Recently Used* (MRU), το οποίο κατανέμει εργασίες στον πιο πρόσφατα ενεργό εργάτη που είναι διαθέσιμος, αν και διακινδυνεύει την λιμοκτονία κόμβων, καθώς ορισμένοι κόμβοι μπορεί να παραμείνουν υποχρησιμοποιούμενοι · v) *Best Effort* (BE), το οποίο ακολουθεί μια άπληστη πολιτική, εξετάζοντας από τον ισχυρότερο κόμβο προς στον ασθενέστερο μέχρι να βρεθεί ένας διαθέσιμος, και στη συνέχεια αναθέτει την εργασία σε αυτόν τον κόμβο-εργάτη για εκτέλεση. Επιπλέον, υλοποιούμε από την αρχή και συγκρίνουμε με μια state-of-the-art λύση χρονοπρογραμματισμού που προέρχεται από το [1]. Εστιάζουμε στην προτεινόμενη λύση χωρίς τεμαχισμό μοντέλου, με την ονομασία SLO Minimum-Average-Expected-Latency (SLO-MAEL), καθώς η εργασία μας δεν επικεντρώνεται στις τεχνικές τεμαχισμού DNN. Το SLO-MAEL στοχεύει στη μείωση των παραβιάσεων QoS και η λήψη αποφάσεων βασίζεται στην αξιολόγηση όλων των πιθανών αντιστοιχίσεων των μηχανών πρόβλεψης σε κόμβους-εργάτες χρησιμοποιώντας ένα σύστημα βαθμολόγησης. Η απόδοση κάθε συστήματος χρονοπρογραμματισμού αξιολογείται χρησιμοποιώντας τις ακόλουθες μετρικές: α) συνολικός αριθμός παραβιάσεων, β) χρόνος αναμονής, ο οποίος αναφέρεται στη διάρκεια που η μηχανή πρόβλεψης περνά στην ουρά, γ) χρόνος εκτέλεσης από άκρο σε άκρο, ο οποίος περιλαμβάνει τόσο τους χρόνους αναμονής όσο και τους χρόνους εκτέλεσης, και δ) μέσος χρόνος υπέρβασης, ο οποίος μετρά τον μέσο χρόνο που μια εργασία υπερβαίνει τον επιθυμητό χρόνο εκτέλεσής της. Αυτό υπολογίζεται ως η διαφορά μεταξύ των πραγματικών και των επιθυμητών χρόνων, με τον χρόνο υπέρβασης να περιορίζεται στο μηδέν για εργασίες που δεν παραβιάζουν το QoS.

4.2 Λεπτομερής Ανάλυση του SynergAI έναντι Καθορισμένων Σημείων Αναφοράς

Η Εικόνα 1.6 παρουσιάζει τα αποτελέσματα του πειράματος DL-FL. Αναλύοντας τον αριθμό των παραβιάσεων, ο χρονοπρογραμματιστής SRR καταγράφει τον υψηλότερο αριθμό στις 18, ακολουθούμενος από το LRU με 14 και το RR με 11. Ο χρονοπρογραμματιστής MRU υφίσταται 5 παραβιάσεις, ενώ το BE αναφέρει 3. Το SynergAI επιτυγχάνει τις λιγότερες παραβιάσεις, με μόνο 2 καταγεγραμμένες. Πέρα από τον αριθμό παραβιάσεων, το SynergAI επιδεικνύει επίσης ανώτερη απόδοση στον χρόνο εκτέλεσης από άκρο σε άκρο. Η κατανομή του χρόνου εκτέλεσής του κυμαίνεται από ένα ελάχιστο 7,4 δευτερολέπτων έως ένα μέγιστο 4,3 λεπτών, με μέσο όρο 1,24 λεπτά και 99%-ο εκατοστημόριο 4,3 λεπτά. Η μετρική του

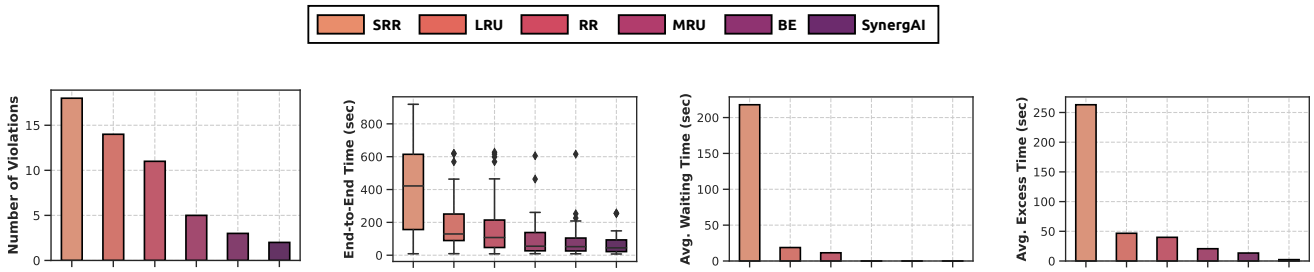


Figure 1.6: Σύγκριση του SynergAI με άλλα Συστήματα Χρονοπρογραμματισμού στο Πείραμα DL/FL

99%-ου εκατοστημορίου είναι ιδιαίτερα σημαντική, καθώς αποτυπώνει τον χρόνο εκτέλεσης της χειρότερης περίπτωσης για το πιο αργό 1% των εργασιών, διασφαλίζοντας την αξιοπιστία του συστήματος υπό μέγιστα φορτία. Το SynergAI ξεπερνά με συνέπεια όλους τους άλλους χρονοπρογραμματιστές, με τον χρόνο εκτέλεσης 99%-ου εκατοστημορίου να είναι χαμηλότερος κατά παράγοντες $2,1\times$, $2,3\times$, $2,4\times$, $2,4\times$ και $3,6\times$ σε σύγκριση με το BE, το MRU, το RR, το LRU και το SRR, αντίστοιχα. Ο μέσος χρόνος αναμονής παρέχει περαιτέρω πληροφορίες σχετικά με την αποδοτικότητα του χρονοπρογραμματισμού. Όπως αναμενόταν, ο χρονοπρογραμματιστής SRR παρουσιάζει τον μεγαλύτερο χρόνο αναμονής στα 3,6 λεπτά λόγω της αυστηρής πολιτικής εναλλαγής εργασιών του, η οποία κατανέμει ομοιόμορφα τους φόρτους εργασίας αλλά αυξάνει τις καθυστερήσεις και τους κινδύνους παραβίασης. Οι χρονοπρογραμματιστές LRU και RR δείχνουν μικρότερους χρόνους αναμονής 18,8 δευτερολέπτων και 11,5 δευτερολέπτων, αντίστοιχα. Αξιοσημείωτα, τόσο το MRU όσο και το BE, μαζί με το SynergAI, δεν παρουσιάζουν καθόλου χρόνο αναμονής. Το πλεονέκτημα του SynergAI προέρχεται από την ικανότητά του να εντοπίζει τη βέλτιστη διαμόρφωση για κάθε μηχανή πρόβλεψης σε όλους τους κόμβους-εργάτες. Αυτό υπογραμμίζει τη σημασία της Offline Phase, καθώς άλλοι χρονοπρογραμματιστές βασίζονται σε προκαθορισμένες διαμορφώσεις, επιλέγοντας συνήθως τον εργάτη με τους υψηλότερους πόρους CPU. Αξιοποιώντας τη γνώση που έχει αποκτήσει, το SynergAI ξεπερνά αυτές τις συμβατικές προσεγγίσεις. Επιπλέον, λόγω της προσαρμοστικής στρατηγικής διαμόρφωσης, ακόμη και για τις δύο εργασίες που υπερέβησαν τους στόχους QoS τους, ο μέσος χρόνος υπέρβασης παραμένει εξαιρετικά χαμηλός στα μόλις 2,3 δευτερόλεπτα. Αυτό είναι σημαντικά χαμηλότερο σε σύγκριση με τους χρόνους υπέρβασης που καταγράφονται από άλλους χρονοπρογραμματιστές: 13,4 δευτερόλεπτα για το BE, 20,7 δευτερόλεπτα για το MRU, 39,9 δευτερόλεπτα για το RR, 46,7 δευτερόλεπτα για το LRU και 4,3 λεπτά για το SRR.

Συνοψίζοντας όλα τα πειράματα, το SynergAI επιτυγχάνει μια μέση μείωση $7,1\times$ στις παραβιάσεις QoS και $5,3\times$ στον χρόνο υπέρβασης, αντίστοιχα.

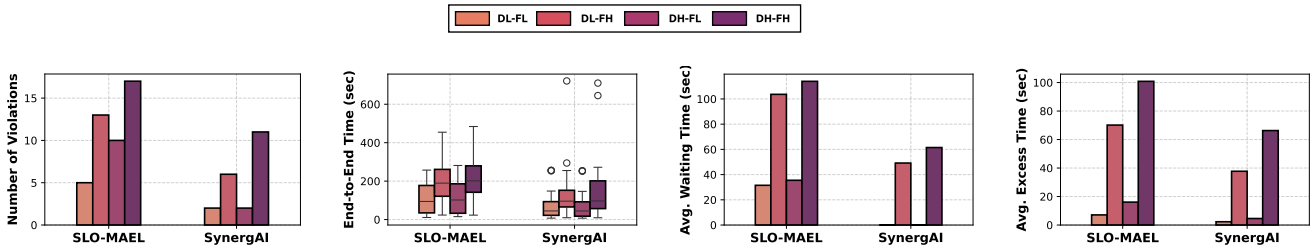


Figure 1.7: Σύγκριση του SynergAI με το SLO Minimum-Average-Expected-Latency (SLO-MAEL) [1] για όλα τα Πειράματα

4.3 Σύγκριση με state-of-the-art Σύστημα Χρονοπρογραμματισμού

Συγκρίνουμε επίσης το SynergAI με το state-of-the-art σύστημα χρονοπρογραμματισμού που προτάθηκε από το [1] (SLO-MAEL). Συγκρίνουμε και πάλι το SynergAI με το SLO-MAEL σε τέσσερα διακριτά πειράματα που αναλύσαμε προηγουμένως: DL-FL, DL-FH, DH-FL, και DH-FH, καθένα αντιπροσωπεύοντας σενάρια με κλιμακούμενη δυσκολία και φόρτο. Για αυτή την αξιολόγηση, χρησιμοποιούμε τις ίδιες μετρικές όπως στην προηγούμενη ανάλυσή μας για να διασφαλίσουμε τη συνέπεια. Η Εικόνα 1.7 παρουσιάζει τα αποτελέσματα από όλα τα πειράματα που διεξήχθησαν.

Στο πείραμα DL-FL, το SLO-MAEL οδηγεί σε 5 παραβιάσεις. Ο χρόνος εκτέλεσης από άκρο σε άκρο ποικίλλει σημαντικά, κυμαινόμενος από ένα ελάχιστο 10,7 δευτερολέπτων έως ένα μέγιστο 4,4 λεπτών, με μέσο όρο 1,8 λεπτά και 99%-ο εκατοστημόριο 4,2 λεπτά. Αντίθετα, όπως έχει ήδη δειχθεί, το SynergAI οδηγεί σε μόνο 2 παραβιάσεις ενώ επιτυγχάνει μείωση $1,5\times$ στο μέσο χρόνο εκτέλεσης. Αυτό το χάσμα απόδοσης μπορεί να αποδοθεί κυρίως στο χρόνο αναμονής που βιώνουν οι εργασίες. Το SLO-MAEL παρουσιάζει μέσο χρόνο αναμονής 31,5 δευτερολέπτων, ενώ το SynergAI εξαλείφει εντελώς το χρόνο αναμονής, επιτρέποντας πιο αποδοτικό χρονοπρογραμματισμό και εκτέλεση. Αν και το SLO-MAEL χρησιμοποιεί ένα σύστημα βαθμολόγησης για τη βελτιστοποίηση της τοποθέτησης εργασίας-εργάτη με βάση τις τρέχουσες συνθήκες ουράς, στερείται προσαρμοστικών δυνατοτήτων επαναχρονοπρογραμματισμού και αποτυγχάνει να λάβει υπόψη τις βέλτιστες διαμορφώσεις πόρων κάθε κόμβου-εργάτη. Κατά συνέπεια, αυτό οδηγεί σε υψηλότερο αριθμό παραβιάσεων και μεγαλύτερους χρόνους αναμονής. Τα μειονεκτήματα του SLO-MAEL είναι επίσης εμφανή στο μέσο χρόνο υπέρβασης εκτέλεσης, ο οποίος μετρά πόσο περισσότερο διαρκούν οι εργασίες πέρα από την αναμενόμενη διάρκειά τους. Εδώ, το SLO-MAEL καταγράφει μέσο χρόνο υπέρβασης 7,1 δευτερολέπτων, ο οποίος είναι $3,1\times$ υψηλότερος από αυτόν της προσέγγισής μας. Αυτό υπογραμμίζει περαιτέρω τις ανεπάρκειες στις αποφάσεις χρονοπρογραμματισμού του σε σύγκριση με το SynergAI, το οποίο εξισορροπεί βέλτιστα τις αναθέσεις εργασιών για την ελαχιστοποίηση καθυστερήσεων και παραβιάσεων. Αυτά τα αποτελέσματα δείχνουν σαφώς ότι το SynergAI ξεπερνά το SLO-MAEL ως προς την αποδοτικότητα χρονοπρογραμματισμού μειώνοντας τις

παραβιάσεις, τους χρόνους εκτέλεσης, τους χρόνους αναμονής και την υπέρβαση εκτέλεσης, τοποθετώντας το ως μια πιο αποτελεσματική λύση για χρονοπρογραμματισμό προβλέψεων.

5 Συμπεράσματα και Επόμενα Βήματα

5.1 Ανακεφαλαίωση

Σε αυτή την εργασία, παρουσιάζουμε το `SynergAI`, ένα ευφυές πλαίσιο χρονοπρογραμματισμού σχεδιασμένο για τη δυναμική βελτιστοποίηση της κατανομής φόρτου εργασίας σε ετερογενή περιβάλλοντα edge και cloud. Αξιοποιώντας εκτενή χαρακτηρισμό επιδόσεων, το `SynergAI` κατανέμει αποτελεσματικά φόρτους εργασίας εξυπηρέτησης προβλέψεων για την ελαχιστοποίηση παραβιάσεων QoS, μεγιστοποιώντας παράλληλα την αξιοποίηση των πόρων. Το πλαίσιό μας λαμβάνει υπόψιν τους συμβιβασμούς μεταξύ επιδόσεων και καταστάσεων λειτουργίας της αρχιτεκτονικής, διασφαλίζοντας μια ισορροπημένη στρατηγική ανάπτυξης σε όλο το υπολογιστικό συνεχές. Μέσω της ομαλής ενσωμάτωσης σε ένα οικοσύστημα βασισμένο στο Kubernetes, το `SynergAI` αποδεικνύει την αποτελεσματικότητά του στη διαχείριση διαφορετικών σεναρίων εξυπηρέτησης προβλέψεων, προσαρμοζόμενο σε μεταβαλλόμενους φόρτους εργασίας και βελτιώνοντας τη συνολική αποδοτικότητα του συστήματος. Τα ευρήματά μας δείχνουν ότι η εξυπηρέτηση προβλέψεων με γνώμονα την αρχιτεκτονική διευκολύνει τις βελτιστοποιημένες, αποδοτικές αναπτύξεις σε αναδυόμενες πλατφόρμες υλικού, με αποτέλεσμα μια μέση μείωση κατά 2.4× στις παραβιάσεις QoS σε σύγκριση με μια state-of-the-art λύση.

5.2 Επόμενα Βήματα

Για μελλοντική εργασία, σχεδιάζουμε να επεκτείνουμε το `SynergAI` ώστε να ενσωματώσει επιτάχυνση GPU σε όλο το edge-to-cloud συνεχές, γεγονός που θα ενισχύσει σημαντικά την απόδοση για υπολογιστικά απαιτητικές εργασίες πρόβλεψης DNN. Επιπλέον, στοχεύουμε να ενσωματώσουμε την δυνατότητα διαμερισμού DNN στο πλαίσιό μας, επιτρέποντας στο σύστημα να διαχωρίζει και να κατανέμει αυτόματα τα επίπεδα νευρωνικών δικτύων σε συσκευές edge με βάση τις υπολογιστικές τους δυνατότητες και τον τρέχοντα φόρτο. Μια τέτοια προσέγγιση θα μειώσει περαιτέρω την καθυστέρηση και την κατανάλωση πόρων βελτιστοποιώντας τόσο το πού όσο και το πώς εκτελούνται οι εργασίες προβλέψης. Συνδυάζοντας την επιτάχυνση GPU με τον ευφυή διαμερισμό DNN, το SynergAI θα μπορούσε να επιτύχει ακόμη μεγαλύτερες μειώσεις στις παραβιάσεις QoS, διατηρώντας παράλληλα την ενεργειακή αποδοτικότητα σε ετερογενή περιβάλλοντα edge-cloud.

Chapter 2

Introduction

In recent years, the rapid advancement of AI and ML applications and their widespread integration into daily life has brought the new era of intelligent computing. Technologies such as Deep Learning (DL) [26], Large Language Models (LLMs) [27], and Dynamic ML [28] are being widely used for inference across diverse domains, including healthcare [29], computer vision [30], and finance [31]. Modern ML inference is characterized by the convergence of increasingly complex models, rapid adaptation to specialized domains, and growing end-user demands. This combination results in significantly higher computing, memory, and storage requirements. For example, SotA DL models used by millions of people daily, such as GPT-3 [2] and DeepSeek-V2 [3], contain 175 billion and 236 billion parameters, respectively. As a result, they impose substantial resource demands and risk performance bottlenecks.

Traditionally, these models are governed by throughput- and latency-oriented QoS [4] requirements, as well as Service Level Objectives (SLOs) or Service Level Agreements (SLAs) [5]. To meet these QoS demands and achieve SLO targets, such models are typically deployed on high-performance Cloud computing platforms, which help mitigate performance bottlenecks while offering scalable, flexible, and cost-effective solutions. Modern Cloud systems enhance ML/DL workloads by utilizing distributed computing, specialized hardware accelerators (e.g., GPUs, TPUs) [6], and optimized networking. Many vendors provide Machine Learning as a Service (MLaaS), offering comprehensive solutions for model training, deployment, and inference. Notable platforms include Google Cloud Vertex AI [7], AWS SageMaker [8], Microsoft Azure ML [9], IBM Watson ML [10], and Nvidia AI Enterprise [32]. However, as service demands continue to surge, Cloud congestion becomes an increasing challenge, leading to limitations in scalability and reliability. Additionally, Cloud servers and data centers are typically centralized and located far from end-user devices. As a result, latency-sensitive applications often suffer from long round-trip delays, network congestion, and degraded service quality [33]. Furthermore, in data-sensitive sectors such as healthcare, storing sensitive information on third-party

infrastructure raises critical privacy and confidentiality concerns [34].

To overcome the limitations of Cloud infrastructures, some of the computational workload is shifted to the Edge, bringing processing closer to where the data is generated. Specialized AI hardware, such as Nvidia Jetson, Google Edge TPU, and Intel Movidius [35], along with optimized ML frameworks like ONNX Runtime, TensorFlow, and TensorFlow Lite [21, 19, 20], enable efficient AI inference directly on Edge devices. This advancement allows for low-power, high-performance AI applications, reducing the need for constant Cloud connectivity and shaping the modern paradigm of Edge AI [11]. Additionally, Edge processing improves power efficiency by reducing reliance on energy-intensive Cloud data centers, lowering data transmission costs, and utilizing specialized AI accelerators [36]. However, Edge computing operates with more limited computational and storage resources compared to Cloud-based processing. As a result, achieving effective Edge-to-Cloud synergy is crucial for enabling scalable and efficient AI inference across the computing continuum.

A major challenge in achieving seamless Edge-to-Cloud synergy is integrating the QoS guarantees of the Cloud with the efficiency, low overhead, and advantages of the Edge while mitigating the inherent limitations of each processing layer. This requires addressing several key challenges: **i) Edge-to-Cloud Scheduling & Heterogeneity:** Effective collaboration between Edge and Cloud depends on intelligent scheduling of inference workloads while accounting for the diverse architectures of modern processors, such as ARM, RISC-V, and x86. **ii) Architecture-Aware Inference Deployment:** Many Edge platforms offer configurable power and performance modes (e.g., power modes, turbo boost) that introduce trade-offs between energy efficiency and inference speed. Optimal deployment requires dynamic tuning of execution settings to align with workload demands. Moreover, the growing diversity in CPU architectures—including differences in core counts, instruction sets (e.g., ARM, x86, RISC-V), and power-performance trade-offs further complicates optimized inference scheduling and resource allocation. **iii) Diverse QoS and SLA Requirements:** Different application domains impose varying QoS constraints. For example, healthcare and automotive applications require millisecond-level inference latency, whereas finance and agriculture can tolerate higher variability, enabling flexible resource allocation across the Edge-to-Cloud continuum. **iv) Inference Engine Variability:** The broad range of inference-serving frameworks, such as TensorFlow [19], ONNX Runtime [21], and PyTorch [37], introduces complexity in selecting the optimal model and backend for deployment across different hardware nodes. The aforementioned challenges form a multi-dimensional problem for the efficient deployment of inference-serving workloads within a heterogeneous Edge-to-Cloud continuum.

Efficient inference serving remains a challenging task, as it requires a deep understanding of application architecture, workload characteristics, and the processing capabilities of both Edge and Cloud infrastructures [4]. To address these challenges, prior research has explored inference serving and orchestration solutions specifically tailored for the Edge [12, 13, 38, 39], the Cloud [4, 14, 40], and the broader Edge-to-Cloud continuum [15, 16, 41, 42]. SotA research offers effective inference-serving strategies, primarily focusing on QoS-driven performance optimizations [43], which serve as the core optimization objective of this work. However, existing approaches lack architecture-aware scheduling frameworks that dynamically optimize inference serving by adapting to the diverse operating capabilities of heterogeneous Edge and Cloud nodes. Additionally, while various scheduling techniques have been proposed for individual layers (Edge or Cloud) and across the continuum, they often fail to fully exploit dynamic resource adaptation and architecture-driven execution modes. As a result, these limitations restrict the optimization and customization potential of existing solutions, preventing them from maximizing efficiency under varying workload demands.

In this work, we introduce **`SynergAI`**, a novel framework designed for architecture-tuned, performance-efficient inference serving across heterogeneous Edge-to-Cloud nodes. The primary optimization goal of `SynergAI` is to minimize QoS violations for inference engines deployed across the continuum by ensuring efficient, architecture-driven engine placement. Our solution is based on comprehensive performance and architecture-driven characterization and analysis of discrete inference engines across different Edge/Cloud nodes and operating modes. Specifically, we focus on the performance and resource trade-off capabilities of Edge nodes, enabling the development of optimized deployment strategies tailored to each node's unique attributes. The outcome of this analysis provides the crucial tuning parameters for our proposed Edge-to-Cloud orchestrator. `SynergAI` utilizes a QoS-aware, architecture-driven scheduling framework that dynamically adjusts the placement of inference jobs based on real-time assessment of QoS violation risks. To summarize, the key contributions of this work are as follows:

- We conduct an extensive **characterization and analysis** for performance and architecture-driven tuning and deployment of discrete ML inference engines and models across the Edge-Cloud continuum.

- **We present `SynergAI`, a novel Edge-to-Cloud scheduling framework** for solving the problem of QoS violations minimization. We incorporate a combination of offline and online mechanisms into our proposed solution, which leverages the characterization and analysis process to per-

form dynamic task scheduling based on real-time assessments of QoS violation risks.

- **We integrate and evaluate our solution with the Kubernetes framework**, demonstrating that `SynergAI`'s architecture-driven inference serving enables optimized and architecture-aware deployments on emerging hardware platforms, achieving an average reduction of 2.4× in QoS violations compared to a SotA solution.

The rest of this thesis is organized as follows. Section 3 presents an overview of the related work. Section 4 provides basic background information about Kubernetes and Edge Computing fundamentals. In Section 5 we provide an extensive characterization and analysis of discrete ML inference engines and nodes, while in Section 6 we present `SynergAI`'s architecture. In Section 7 the experimental evaluation is presented and discussed. Finally, Section 8 concludes this research and suggests directions for future work.

Chapter 3

Related Work

In recent years, numerous studies have focused on inference serving and scheduling. Systems like TensorFlow Serving [19], TensorFlow Lite [20], and ONNX Runtime [21] are notable examples of flexible and high-performance serving platforms for ML models, tailored for production environments on both Edge and Cloud infrastructures. From an industrial perspective, Nvidia's AI platform offers the Triton Inference Server [44], which facilitates GPU-based inference while also supporting CPU models, though it requires static configuration for model instances. Regardless of the setup, effective inference serving relies heavily on efficient scheduling and orchestration. In this section, we present related work, categorized based on the target optimization layer within the Edge-to-Cloud continuum. Specifically, we group the examined research into three main categories: (i) *Inference Serving & Scheduling on the Edge*, (ii) *Inference Serving & Scheduling on the Cloud*, and (iii) *Inference Serving & Scheduling across the Edge-Cloud Continuum*.

Inference serving & Scheduling on the Edge: Several studies have focused on optimizing inference serving at the Edge. In [12], the authors propose a DNN partitioning and offloading approach tailored for Edge computing systems. Similarly, in [38], a framework is presented that utilizes edge computing for collaborative DNN inference through device-edge synergy. In [39], a mathematical model is introduced for adaptive DNN model partitioning and inference offloading at the Edge. Additionally, [45] proposes a multi-agent reinforcement learning-based, energy-efficient collaborative inference scheme for Mobile Edge Computing (MEC). Lastly, in [13], a resource-efficient DNN inference method with latency-awareness is proposed, while [46] explores the trade-offs between energy consumption and latency for CNN inference on Edge accelerators.

Inference Serving & Scheduling on the Cloud: Several studies have investigated inference serving, scheduling, and orchestration at the Cloud layer. In [4], the authors introduce an interference- and resource-aware predictive orchestrator for ML inference serving. In [14, 40], a scheduling approach for CPU servers is presented, which utilizes load prediction to reduce interference.

Furthermore, [47] proposes a modular framework that balances incoming workloads based on low-level metrics monitoring. Resource partitioning strategies designed to optimize QoS requirements have also been explored in [48]. Finally, in [49], workload-specific scheduling techniques are discussed, where different workload classes are handled by distinct schedulers.

Inference Serving & Scheduling across the Edge-Cloud Continuum: To leverage the advantages of both Edge and Cloud computing, numerous studies have focused on optimizing inference serving and scheduling across the Edge-Cloud continuum. In [15], the authors propose an active inference-based approach for offloading LLM inference tasks and managing resource allocation in cloud-edge environments. The study in [50] introduces an Edge-Cloud collaborative architecture for DNN inference, while [16] presents a preemptive scheduling solution for distributed ML jobs in Edge-Cloud networks. Additionally, [51] employs LLMs to dynamically adjust the placement of application tasks across Edge and Cloud layers in response to workload fluctuations. In [41], the authors propose an offloading scheme to accelerate DNN inference in a local-edge-cloud collaborative setting, and [42] presents a deep reinforcement learning-based strategy for optimizing DNN offloading across Edge and Cloud environments.

While several approaches to inference serving and scheduling across the Edge, Cloud, and Edge-Cloud continuum have been explored in existing research, to the best of our knowledge, no study has proposed an architecture-driven scheduling framework that dynamically adapts to the heterogeneous capabilities of Edge and Cloud nodes to optimize inference serving. Our system takes advantage of the architecture-driven performance features and operating modes of modern Edge/Cloud infrastructures, leveraging their configurable modes to enhance efficiency and minimize QoS violations. Through extensive architecture-driven characterization and analysis, our end-to-end orchestrator intelligently allocates ML workloads across the continuum, offering a lightweight and performance-optimized solution.

Chapter 4

Background Fundamentals

1 The Kubernetes Orchestrator

1.1 Virtualization

Virtualization is a process that creates digital environments that function independently from the physical hardware they run on. This technology enables a single physical machine to host multiple virtual instances, each operating as if it were a standalone device. These virtual instances can serve various purposes, from complete computing environments to specific applications or network configurations. The process works through a specialized software called a hypervisor, which functions as an intermediary layer between the physical hardware and the virtual environments. This lightweight but powerful software manages resource allocation, allowing several operating systems to function simultaneously on the same physical hardware. The hypervisor effectively distributes computing resources from the underlying physical infrastructure to each virtual instance as needed.

1.2 Virtual Machines

Virtual machines (VMs) are the digital environments created through virtualization technology, functioning as independent computing systems despite sharing the same physical hardware. Each VM needs its own operating system and accesses hardware resources from either a single host server or a distributed server pool. The hypervisor creates this virtual hardware layer while maintaining isolation between different VMs. IT departments have widely adopted VM technology because it offers cost savings and operational efficiencies compared to maintaining separate physical machines. However, VMs consume significant system resources since each one requires not only a complete operating system but also virtualized versions of all necessary hardware components. These self-contained units, frequently used to run large, single-file applications, require substantial memory and processing power. The considerable resource require-

ments of VMs, while still more economical than dedicated physical machines, can be excessive for certain applications. This limitation ultimately inspired the development of more lightweight container technology.

1.3 Containers and Container Runtimes

Containers represent a lightweight virtualization approach that fundamentally differs from traditional virtual machines. While VMs virtualize the entire computer system, containers virtualize only the operating system layer. Containers operate directly on the host system's OS—typically Linux or Windows—sharing the kernel, binaries, and libraries with read-only access to these shared components. This resource-sharing architecture significantly reduces duplication, enabling a single operating system installation to efficiently support multiple isolated workloads. Containers offer remarkable efficiency advantages in both size and speed, as they typically occupy megabytes of space compared to gigabytes for VMs and initialize in seconds rather than minutes. This translates to substantially higher deployment density, with servers commonly supporting 2-3 times more containerized applications than equivalent VM-based deployments. In this thesis, our SynergAI system utilizes `containerd` as its container runtime. `Containerd` is an industry-standard container runtime that manages the complete container lifecycle, from image transfer and storage to container execution and supervision. Originally developed by `Docker`, `containerd` was later donated to the Cloud Native Computing Foundation (CNCF) to serve as an independent, stable foundation for container runtimes. As the default runtime for Kubernetes since version 1.20, `containerd` offers a minimal yet powerful interface for container operations, optimized performance, and robust security features. While `containerd` serves as our container runtime in `SynergAI`, several other container runtime options exist within the container orchestration landscape. `Docker Engine` remains a popular alternative that provides additional developer-friendly features and tools layered on top of `containerd`. Other options include `CRI-O`, a lightweight runtime designed specifically for Kubernetes, and `rkt` (rocket), which emphasizes security and composability.

1.4 Container Orchestration and Kubernetes

As container adoption grows within organizations, managing individual containers at scale becomes increasingly complex. Container orchestration addresses this challenge by automating the deployment, management, scaling, networking, and availability of containerized applications across distributed systems. While containers provide efficient application packaging and isolation, orchestration coordinates these containers to function as cohesive systems. Kubernetes has emerged as the leading container orchestration platform, offering a portable,

extensible solution for managing containerized applications. As an open-source project since 2014, it provides a configuration-driven approach to infrastructure management that abstracts away underlying complexity, enabling organizations to focus on application development rather than operational challenges. In this thesis, we utilize Kubernetes [17] (v1.28.10) to create and manage our experimental cluster, and the following paragraph will explain its core components and objects in detail.

1.5 Kubernetes Architecture and Resource Types

1.5.1 Cluster

A Kubernetes cluster represents the highest-level abstraction in the Kubernetes ecosystem, consisting of a collection of machines, physical or virtual, that work together to run containerized applications. This group of interconnected machines provides the infrastructure upon which Kubernetes operates as a powerful orchestration system. The cluster follows a master-worker architecture, with nodes separated into two distinct groups: master nodes that form the control plane and worker nodes where actual workloads are executed, as illustrated in Figure 4.1.

1.5.2 Master-Worker Architecture

The master nodes collectively form the control plane, that acts as the brain of the cluster, making global decisions and maintaining the desired state of the system. A Kubernetes cluster must have at least one master node, which commands and controls all other machines in the cluster. Key components include the API server which serves as the central communication hub, etcd that maintains cluster state in a distributed key-value store, the scheduler which is responsible for workload assignment, and various controller managers that regulate system state.

Worker nodes provide the execution environment where containerized applications actually run. Each worker node operates essential components: the kubelet agent that communicates with the control plane, kube-proxy for networking services, and a container runtime that executes the containers themselves.

This architectural separation creates a clear distinction of responsibilities, with master nodes handling cluster-wide control and coordination while worker nodes focus on executing the workloads. In highly available setups, the master's capabilities can be replicated across multiple machines for fault tolerance, though typically only one master actively runs critical components like the scheduler and controller manager at any given time.

1.5.3 Core Objects and Concepts

Kubernetes organizes workloads through several fundamental objects:

Pods are the most basic deployable units in Kubernetes, with each pod hosting one or more containers that share network and storage resources. They represent single instances of applications or running processes in Kubernetes, functioning as the smallest execution units that can be created and managed. To keep the focus on the application rather than individual containers, Kubernetes manages pods as cohesive units, starting, stopping, and replicating all containers within a pod together. By design, pods are ephemeral, dynamically created and destroyed as necessary to maintain the desired state of the system.

Deployments provide declarative definitions for applications, specifying the desired state that Kubernetes works to maintain. The Deployment controller ensures the actual state matches this specification, handling scaling, updates, and rollbacks at a controlled rate. Deployments create and manage ReplicaSets, which are lower-level resources that maintain a stable set of replica pods running at any given time. The hierarchical structure enables powerful capabilities like rolling updates and automated recovery from failures. This approach offers straightforward scalability by simply adjusting the number of pod replicas, allowing applications to easily adapt to changing demands without service disruption.

Jobs are controller objects that create and manage pods designed to run until their processes complete successfully. Unlike Deployments, which manage continuously running applications, Jobs are ideal for batch processing or one-time tasks that must run to completion, ensuring reliable execution of workloads with a finite lifespan. When a Job is created, it spawns one or more pods that execute until they finish their assigned work, at which point they terminate instead of restarting. Jobs belong to the same family of controller objects as Deployments but serve a distinctly different purpose in the Kubernetes ecosystem, handling tasks with a clear beginning and end rather than maintaining persistent services.

[1] https://kubernetes.io/docs/concepts/architecture/

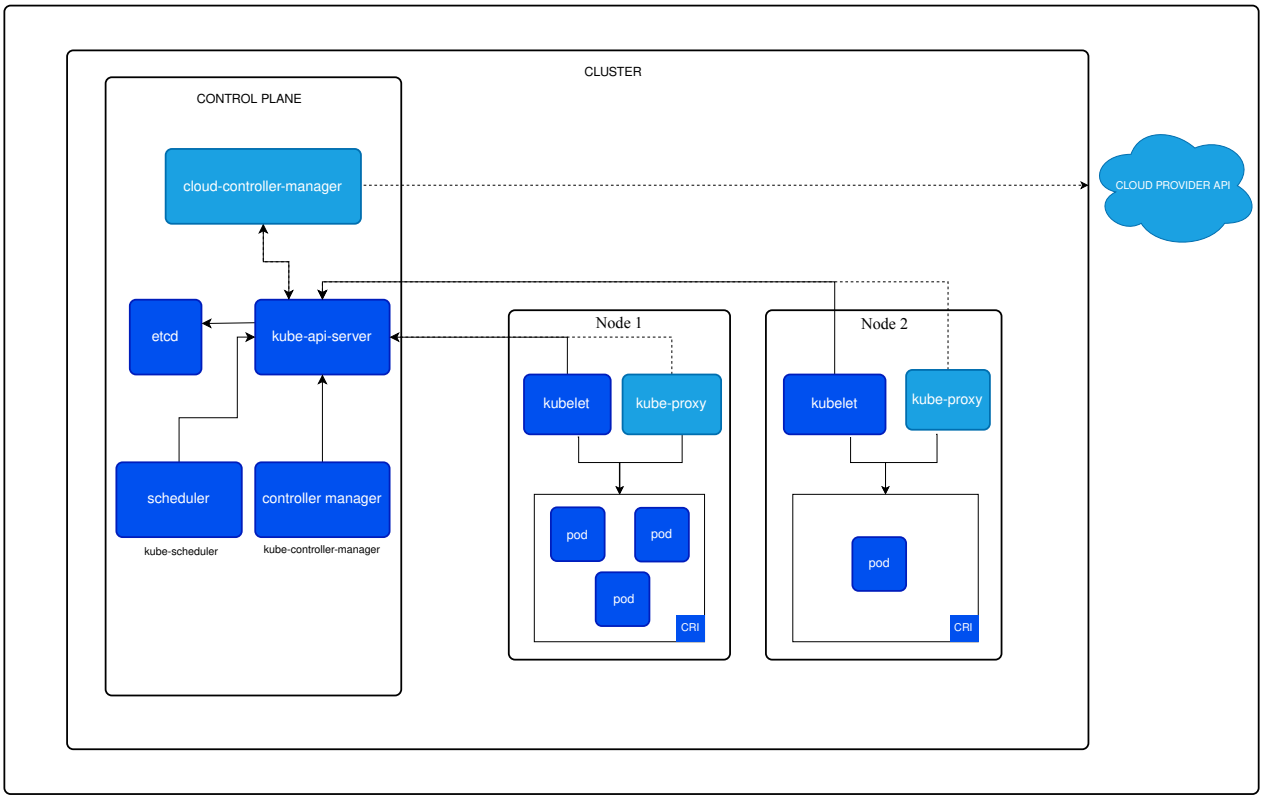


Figure 4.1: Kubernetes Cluster Architecture [1]

2 Edge Computing

2.1 Edge Computing Fundamentals

Edge computing brings resources closer to data sources, enabling processing and analysis near the network edge instead of relying on distant cloud data centers. This approach marks a shift from centralized computing to processing at the network endpoints [52]. Resources are distributed throughout the network, creating an architecture of computing nodes near users and data sources. This distribution helps support mobile users by providing seamless service as they move between different edge nodes. The diverse range of devices, platforms, and networks in edge computing creates both challenges and opportunities for developers [35]. Several edge computing models have emerged for different uses. Cloudlets are small-scale data centers at the network edge that support mobile devices in businesses or public spaces. Fog Computing creates a distributed computing structure for IoT applications. Mobile Edge Computing brings cloud capabilities into mobile networks. Micro Data Centers are compact, modular facilities that can be deployed at the network edge for industrial applications [52].

2.2 Edge vs. Cloud Computing Differences

Edge computing differs from cloud computing in several key aspects. While cloud computing centralizes processing in distant data centers, edge computing places resources close to data sources and thus significantly reduces latency, which is essential for time-sensitive applications like autonomous vehicles and industrial systems. Edge computing also uses less bandwidth by processing data locally, unlike cloud computing which requires sending all data to and from central data centers. Additionally, edge systems can use information about the physical location of devices and users to provide context-aware services, a capability that traditional cloud architectures struggle to deliver effectively [53]. The scaling approaches also differ between these models. Cloud computing offers virtually unlimited scaling within centralized facilities, while edge computing provides more limited resources at individual nodes but can scale across many distributed locations. Security considerations vary too, with edge computing offering improved security by keeping sensitive data local while simultaneously presenting new challenges due to its distributed nature [35].

2.3 Edge-to-Cloud Continuum

The edge-to-cloud continuum represents a computing approach that uses both edge and cloud resources together. Rather than seeing edge and cloud as competing options, this view presents them as points on a spectrum, creating a seamless computing infrastructure from the network edge to central data centers. This approach allows for more flexible application deployment and resource use, taking advantage of the strengths of each approach [54].

The composition of this infrastructure includes a range of computing resources at different locations. At the furthest edge are devices like IoT sensors, smartphones, and other data-generating equipment. These devices usually have limited computing capabilities but can perform basic processing. Moving inward, the edge layer includes gateways, servers, and base stations that provide more substantial computing, storage, and networking capabilities near data sources. This layer processes and analyzes data locally, reducing the need to send all information to distant facilities [54]. At the center is the traditional cloud layer, with its vast computing and storage resources in data centers. The cloud handles tasks that need significant processing power or global data analysis. This multi-layered structure supports computing from the most appropriate location based on what's needed, what resources are available, and network conditions [55].

The edge-to-cloud continuum offers clear advantages over using either edge or cloud computing alone. It enables better resource use by assigning computing tasks based on their specific requirements: resource-intensive operations can run

in the cloud, while time-sensitive processing happens at the edge. This strategy improves overall system performance, providing lower latency for critical applications while maintaining high throughput for data-intensive workloads [35]. This approach improves scalability by distributing computation across different levels based on demand. When local edge resources are strained, tasks can move to the cloud, and when cloud connectivity is limited, more processing can happen at the edge. The distributed architecture also improves reliability through redundancy, as services can redistribute if specific nodes fail [35].

Deciding where to run computational tasks is a critical aspect of the edge-to-cloud continuum. This decision depends on multiple factors that must be carefully balanced. Time-sensitive tasks with low computing demands typically run at the edge, minimizing delays for critical operations. In contrast, computing-intensive tasks with less strict timing requirements may run in the cloud, using its greater processing capabilities [56]. Network conditions greatly influence these decisions as well. Available bandwidth, connection speed, and network reliability all help determine the best location for specific tasks. Similarly, resource availability affects task placement, as the current load and available computing power at different nodes can shift the balance between edge and cloud execution. For battery-powered devices, energy use is an important consideration, as offloading computation can extend battery life but must be weighed against the energy costs of data transmission [53].

This hybrid architecture enables diverse applications across various domains including smart city monitoring, industrial IoT control systems, and healthcare alerts, with each utilizing edge for time-sensitive processing and cloud for in-depth analytics [56]. Autonomous vehicles represent a compelling application that requires edge processing for immediate obstacle avoidance decisions where milliseconds matter, while benefiting from cloud resources for route planning and traffic optimization using broader data sets. Augmented reality similarly uses edge computing to reduce latency for real-time rendering while leveraging cloud capabilities for complex scene understanding [55].

Despite these benefits, the continuum faces significant challenges. Resource management is complex due to device diversity, while security and privacy protection is difficult across multiple tiers and administrative domains. Standardization is lacking for component integration, and maintaining consistent service quality with varying network conditions requires sophisticated monitoring and adaptation mechanisms. Additionally, the operational complexity of managing this distributed architecture creates substantial implementation barriers for many organizations [54].

Chapter 5

Profiling, Characterization & Analysis

1 Inference Serving Testbed

Hardware & Software Infrastructure: We conduct our experiments on a high-end dual-socket Intel® Xeon® Gold 5218R (@2.1 GHz) server, equipped with 128 GB of DRAM memory. On top of this setup, we configure two virtual machines to function as the master (4 vCPUs, 8 GB RAM) and worker (16 vCPUs, 16 GB RAM) nodes of our cluster, respectively, utilizing the KVM hypervisor. Additionally, we integrate two more worker nodes on the Edge, which include an Nvidia Jetson AGX (8 CPUs, 32 GB RAM) and an Nvidia Jetson Xavier NX (6 CPUs, 8 GB RAM). For cluster deployment and orchestration, we leverage Kubernetes [17] (v1.28.10) in combination with Containerd (v1.7.2) as the container runtime.

Inference Engine Workloads: For the purposes of this thesis, we use the object detection and image classification tasks from the MLPerf Inference benchmark suite [18]. Table 5.1 provides details on the MLPerf inference engines used, including the representation, the model variant, the validation dataset, and the model accuracy. Each inference engine container instance comprises two components: (i) the Inference Engine and (ii) the Load Generator. The Inference Engine processes a pre-trained DNN model (e.g., ResNet) using a specified backend framework (e.g., TensorFlow) to perform the designated task. Meanwhile, the Load Generator simulates traffic for the Inference Engine and monitors its performance. It takes as input the validation dataset (e.g., ImageNet), the scenario, and the total number of queries to execute. In our experiments, we employed the Single Stream scenario, where the Load Generator sends one sample per query and waits for execution to complete before dispatching the next one. Throughout the characterization, we maintain a consistent number of queries in MLPerf across all inference engines to evaluate their performance under the same workload and uncover their internal characteristics. This value is the default used in the Single Stream scenario.

Table 5.1: Supported MLPerf Inference Engines, Model Variant, Dataset and Corresponding Accuracy

Task	Representation	Model Variant	Dataset	Accuracy
Image Classification	**Tensorflow (TF) [19]**	ResNet50	ImageNet	76.456%
		MobileNet	ImageNet	71.676%
		MobileNet Quantized (Q)	ImageNet	70.694%
	TFLite (TFL) [20]	MobileNet	ImageNet	71.676%
		MobileNet Quantized	ImageNet	70.762%
	ONNX Runtime [21]	ResNet50 `opset-8` (OP8)	ImageNet	76.456%
		ResNet50 `opset-11` (OP11)	ImageNet	76.456%
		MobileNet `opset-8`	ImageNet	71.676%
		MobileNet `opset-11`	ImageNet	71.676%
Object Detection	**TensorFlow [19]**	SSDMobileNet	Coco 300	mAP 0.234
	ONNX Runtime [21]	SSDMobileNet `opset-8`	Coco 300	mAP 0.23
		SSDMobileNet `opset-11`	Coco 300	mAP 0.23

Analysis of Operating Modes in AGX and NX Boards: The Nvidia AGX and NX boards provide various operating modes to optimize performance, energy efficiency, and thermal management. These modes enable the system to switch between peak performance for demanding workloads and lower power consumption for enhanced efficiency in energy-constrained environments. Table 5.2 presents a detailed overview of the modes analyzed in our research, as derived from the manufacturer's specifications. It is important to note that both the AGX and NX boards offer additional modes, but enabling them requires a reboot. Since this work focuses on run-time decision making, we avoid using these modes as they introduce overheads which can affect the scheduling system we are developing. Focusing on the Nvidia Jetson AGX, we observe six available operating modes. The CPU frequency ranges from 1200 MHz to 2266 MHz, with the number of active CPUs varying from 2 to 8. Additionally, the power budget ranges from 15 Watts to MAXN, generally indicating no fixed upper limit on power consumption. This enables the system to operate at peak performance while dynamically adjusting power usage according to workload demands and thermal constraints. On the other side, the Nvidia Jetson Xavier NX offers nine available power modes. The CPU frequency here ranges from 1200 MHz to 1900 MHz, with 2 to 6 CPUs active, and the power budget ranges from 10 Watts to 20 Watts.

2 Characterizing Inference Serving

To gain deeper insights into the execution characteristics of the various inference engines listed in Table 5.1, we evaluate the impact of different factors on their performance. Specifically, we profile the engines to assess how i) x86 and ARM-based workers, ii) vertical resource scaling, and iii) operating modes on ARM-based workers influence their performance. At the end of this analysis, we highlight the key insights from our study to determine the optimal modes for

implementation in our scheduling system.

Table 5.2: Power Mode Configurations for Nvidia Jetson AGX and Xavier NX Boards

Board	Mode	Max CPU Frequency (MHz)	Online CPUs	Power Budget (W)
Nvidia Jetson AGX	Mode 1	1200	8	30
	Mode 2	1450	6	30
	Mode 3	1780	4	30
	Mode 4	2100	2	30
	Mode 5	2188	4	15
	Mode 6	2266	8	MAXN
Nvidia Jetson Xavier NX	Mode 1	1200	4	10
	Mode 2	1400	4	15
	Mode 3	1400	4	20
	Mode 4	1400	6	15
	Mode 5	1400	6	20
	Mode 6	1500	2	10
	Mode 7	1900	2	15
	Mode 8	1900	2	20
	Mode 9	1900	4	10

2.1 Performance-aware Characterization

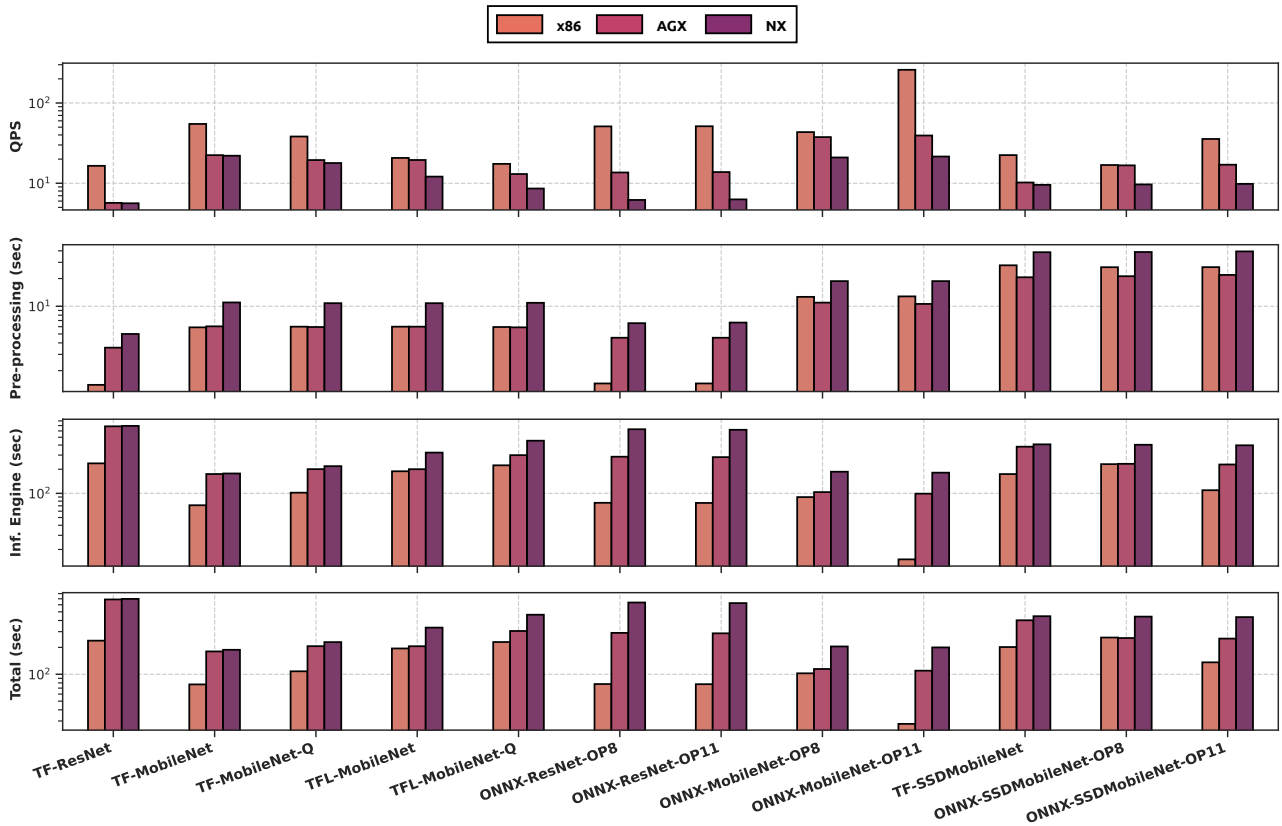


Figure 5.1: Characterization of the Evaluated Inference Engines Across All Available Workers: QPS (Top), pre-processing time (Upper-middle), inference engine time (Lower-middle) and total execution time (Bottom) in log. scale.

In this section, we profile the inference engines listed in Table 5.1 to analyze their performance across all workers in our cluster. Figure 5.1 presents the

characterization of the evaluated inference engines across all available workers. It illustrates (a) the QPS (Top), (b) the pre-processing time, which includes backend initialization, DNN model loading, and validation data preparation (Upper-middle), (c) the inference engine execution time, which refers to the actual computation (Lower-middle), and (d) the total execution time, as recorded by the Load Generator for each MLPerf Inference Engine, which represents the overall pre-processing and inference engine execution time (Bottom). All metrics are depicted in logarithmic scale. For the x86 worker, QPS values range from 16.5 for TensorFlow ResNet to 259 for ONNX Runtime MobileNet with `opset-11`, resulting in an average QPS of 52.3. For total execution time, we observe a distribution ranging from a minimum of 27.8 seconds for ONNX Runtime MobileNet with `opset-11` to a maximum of 4.3 minutes for ONNX Runtime SSDMobileNet with `opset-8`. The average execution time is 2.4 minutes. An interesting observation is that, despite all inference engines processing the same number of queries, the ONNX ResNet engine with the lowest QPS is not the one with the longest execution time. Instead, the shortest execution time is recorded for ONNX Runtime MobileNet with `opset-11`, which has a QPS of 259. This behavior is attributed to the preprocessing time, which includes the overhead of initializing the selected backend, setting up the pre-trained DNN model on this backend, and loading the MLPerf validation dataset. Indeed, TensorFlow ResNet completes this process in just 1.4 seconds, whereas ONNX Runtime MobileNet with `opset-11` requires 19× longer time for preprocessing. This significantly impacts performance, ultimately counteracting the advantage of its higher QPS.

For the AGX worker, QPS ranges from 5.7 for TensorFlow ResNet to 39.3 for ONNX Runtime MobileNet with `opset-11`, with an average of QPS of 19. The total execution time spans from a minimum of 1.8 minutes for ONNX Runtime MobileNet with `opset-11` to a maximum of 11.5 minutes for TensorFlow ResNet, averaging 4.6 minutes. Similarly, for the NX worker, QPS varies between 5.6 for TensorFlow ResNet and 22 for TensorFlow MobileNet, with an average of 12.5. The total execution time ranges from a minimum of 3.1 minutes for TensorFlow MobileNet to a maximum of 11.6 minutes for TensorFlow ResNet, with an average of 6.8 minutes. For both AGX and NX workers, the inference engine with the highest QPS results in the shortest execution time, while the one with the lowest QPS has the longest execution time, unlike the x86 worker. This occurs because preprocessing time increases only slightly, by 10% on AGX and 1.76× on NX compared to x86. Preprocessing tasks like image resizing and normalization rely more on memory bandwidth and I/O rather than CPU power, causing minimal slowdown on weaker hardware. In contrast, inference is far more computationally intensive, making the increase

in preprocessing time negligible compared to the much larger rise in execution time.

⋆ **Q1:** ***How does the performance of inference engines vary when executed on different workers?***

The x86 worker outperforms AGX and NX, achieving 2.8× and 4.2× higher QPS, respectively, while also being 2× and 2.8× faster in execution time, respectively. This is expected, as x86 offers the most powerful CPU and the largest available RAM, followed by AGX and then NX. TensorFlow ResNet consistently exhibits the lowest performance across all devices due to its high computational complexity [22], which becomes even more evident on resource-constrained platforms like AGX and NX. ONNX MobileNet `opset-11` is the fastest inference engine on both x86 and AGX, benefiting from ONNX Runtime's optimizations for efficient parallel execution and low-latency inference. However, on NX, TensorFlow MobileNet achieves the best performance. This analysis highlights the significance of architecture-aware inference deployment, as performance can vary substantially across hardware platforms, even within the same device family, due to the interplay between system architecture, framework optimizations, and the characteristics of the inference model.

⋆ ***Key Outcome 1:*** *Optimal inference engine and model selection varies significantly across different hardware architectures. Inference efficiency is driven by the engine, models and intra-architecture characteristics.*

2.2 Architecture-Driven Tuning Characterization

In this section, we examine the impact of various optimizations on the performance of the analyzed inference engines. Our approach is divided into two parts: a) for the x86 worker, we evaluate how the number of threads influences performance, and b) for the AGX and NX workers, we investigate how different operating modes affect performance.

⋆ **Q2:** ***How does vertical scaling (i.e., #Threads) impact performance on x86-based workers?***

In this part of our analysis, we examine the behavior of various inference engines when executed on an x86 worker with different thread counts. Specifically, for inference engines using ONNX Runtime as the backend, we modify the `INTRA_OP_NUM_THREADS` parameter, and for TensorFlow engines, we adjust the `INTRA_OP_PARALLELISM_THREADS` setting, which are the most influential parameters, as outlined in prior research [4]. Finally, for TensorFlow Lite, we explore the `NUM_THREADS` parameter. Figure 5.2 illustrates the QPS (Top) and execution time (Bottom) across all the inference engines examined (X-axis) for

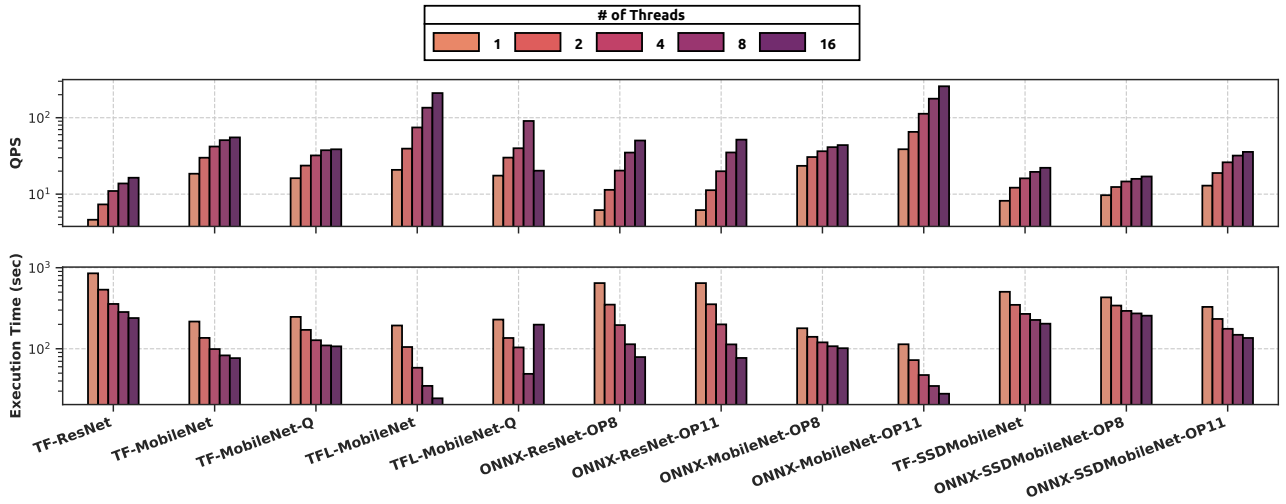


Figure 5.2: Impact of Thread Scaling on MLPerf Inference Engines in x86 Worker

ranging number of threads. As shown in Figure 5.2, increasing the number of available threads results in higher QPS and reduced total execution time. Using 2, 4, 8, and 16 threads yields average QPS improvements of 1.6×, 2.5×, 3.8×, and 4.5× respectively, compared to single-threaded execution. In terms of total execution time, 2, 4, 8, and 16 threads lead to average execution speedups of 1.6×, 2.3×, 2.9×, and 3× compared to single-threaded execution, respectively. This suggests that performance improvements from increasing threads are not always linear or significant beyond a certain point. For instance, while moving from 1 to 8 threads provides a speedup of 2.9×, increasing to 16 threads only yields a marginal improvement to 3×. This diminishing return indicates that beyond a specific threshold, adding more threads does not proportionally enhance performance, likely due to factors such as increased synchronization overhead, contention for shared resources (e.g., cache or memory bandwidth), and inefficiencies in parallel execution. Thus, utilizing all available threads may not always be necessary to achieve near-optimal performance, opening the door for further exploration into the correlation between thread scaling and inference efficiency. Last, the pre-processing time is not significantly affected by the increase in available threads, resulting in a maximum execution time reduction of 21% at 16 threads compared to single-threaded execution. This is expected, as we previously mentioned that pre-processing relies more on memory bandwidth and I/O rather than CPU.

⋆ ***Key Outcome 2:*** *Increasing the number of threads improves inference performance on x86-based workers, but with diminishing returns beyond a specific number of threads, suggesting that near-optimal performance can be achieved without utilizing all available threads.*

⋆ **Q3:** ***Does performance exhibit linear correlation with thread scaling?***

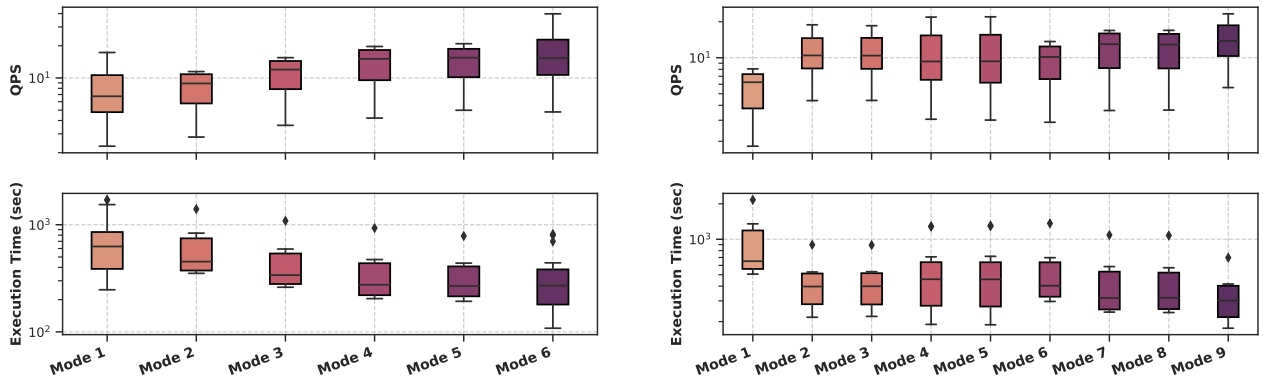
As observed, doubling the number of threads does not lead to a two-fold increase in QPS. Increasing threads beyond a certain point yields diminishing returns due to thread contention and synchronization overheads. As more threads compete for shared resources like main and cache memory, contention increases, leading to reduced efficiency and limited speedup. This effect is even more pronounced in total execution time, where after a certain decrease in inference engine time, preprocessing becomes the dominant factor. For example, in ONNX Runtime SSD MobileNet with `opset-8`, from 4 threads onward, preprocessing dominates, making further increases in thread count negligible in impact.

Another interesting observation is seen in TensorFlow Lite MobileNet Quantized, where performance scales as expected from 1 to 8 threads, but at 16 threads, both QPS and execution time remain nearly identical to single-threaded execution. This behavior is influenced by two key factors. First, TensorFlow Lite's threading model is designed for embedded devices, focusing on low-latency execution rather than aggressive parallelism. As a result, it does not effectively distribute workloads across many threads, particularly on x86 CPUs, which are not its primary target. Beyond 8 threads, the threading model reaches a performance ceiling, causing diminishing returns at 16 threads. To assess the linearity, we calculate the Pearson correlation [23] between execution time and the number of threads, with values closer to 1 indicating a stronger linear correlation. The correlation between the number of threads and performance is 0.83 on average, indicating a strong but not full-linear relationship. Moreover, quantization plays a significant role in inference performance across different hardware platforms. While `int8` computations are generally faster than floating-point operations due to their reduced computational complexity and lower memory bandwidth requirements, the extent of performance gains varies across architectures [24, 25]. Since `int8` operations are inherently lightweight, excessive threading can introduce synchronization overhead, negating potential speedups. Since integer-based operations are lightweight and execute rapidly, the overhead of managing additional threads negates the potential speedup.

⋆ ***Key Outcome 3:*** *Performance does not scale fully linearly with the number of threads due to contention, synchronization overheads, and workload-specific limitations, leading to diminishing returns beyond a certain threshold. Thus, performance over-consumption can be avoided.*

⋆ Q4: *How do operating modes affect ARM-based workers?*

Figure 5.3 illustrates the distributions of QPS and total execution time for the AGX and NX boards across the available operating modes, as outlined in Table 5.2. Focusing on the AGX worker, we observe that Mode 6 stands out



(a) AGX (b) NX

Figure 5.3: Impact of Operating Modes on the QPS (Top) and execution time (Bottom) for the MLPerf Inference Engines for AGX and NX

with the highest QPS and, consequently, the lowest execution time. The QPS distribution ranges from 4.8 to 39.8, averaging at 18.4, while execution time ranges from 1.8 to 13.6 minutes, with an average of 5.1 minutes. Specifically, Mode 6 delivers a QPS that is 1.3× higher than Mode 5, 1.4× higher than Mode 4, 1.8× higher than Mode 3, 2× higher than Mode 2, and 2.2× higher than Mode 1. In contrast, the slowest mode, Mode 1, shows a QPS distribution from 4.8 to 17.3, averaging 8.2, with execution times ranging from 4.1 to 28.5 minutes, averaging 11.4 minutes. Mode 6 is expected to perform the best due to its highest maximum operating frequency, utilization of all available CPUs, and lack of power budget limitations. On the other hand, although Mode 1 offers more online CPUs and a higher power budget, its lower frequency results in it being the slowest option.

For the NX worker, Mode 9 achieves the highest QPS. The QPS distribution ranges from 5.6 to 23.3, averaging 14.3, while execution time varies between 2.9 and 11.6 minutes, with an average of 5.8 minutes. Specifically, Mode 9 delivers a QPS that is 1.2× higher than Modes 7 and 8, 1.3× higher than Modes 2, 3, 4, and 5, 1.5× higher than Mode 6, and 2.5× higher than Mode 1. The lowest-performing mode is Mode 1, with a QPS distribution ranging from 1.8 to 8.1, averaging 5.5, and execution times between 8.4 and 36 minutes, with an average of 16.9 minutes. Mode 9 achieves high performance by utilizing the highest CPU frequency while using only four threads and operating at the lowest power budget. In contrast, Mode 1, which has the lowest maximum CPU frequency of 1200 MHz, along with the same number of CPUs and power budget, results in the slowest execution.

⋆ ***Key Outcome 4:*** *Operating modes significantly impact performance on ARM-based workers, with higher CPU frequencies leading to better QPS and lower execution times.*

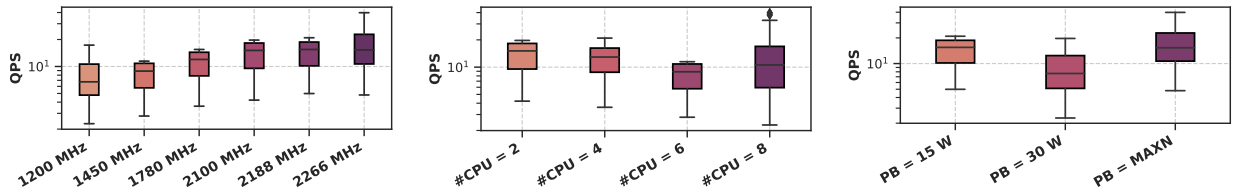


Figure 5.4: Impact of Frequency, #CPUs and Power Budget on Performance for AGX Worker

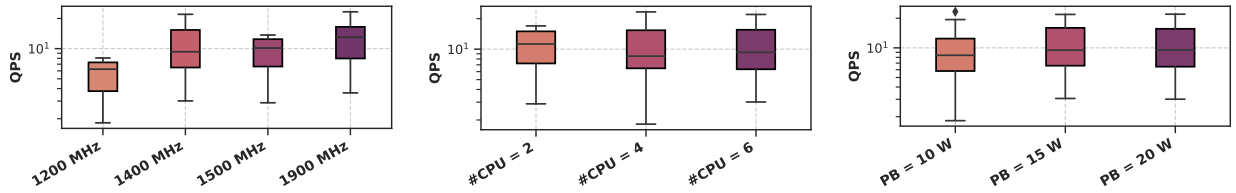


Figure 5.5: Impact of Frequency, #CPUs and Power Budget on Performance for NX Worker

⋆ Q5: *Which parameters (e.g., #CPUs, frequency) have the greatest impact on performance?*

To gain a deeper understanding of why the aforementioned operating modes are significant, we analyze how the performance of the inference engines is affected by the key aspects of each operating mode for both the AGX and NX workers. Figures 5.4 and 5.5 show the distributions of QPS across different maximum operating clock frequencies (Left), the number of online CPUs (Center), and the available power budget (Right). As observed for both AGX and NX workers, increasing the maximum CPU frequency results in higher performance. For the AGX worker, raising the CPU frequency from 1200 MHz to 1450 MHz leads to an average 2.5% increase in QPS. Increasing to 1780 MHz results in a 1.35× boost, to 2100 MHz a 1.7× increase, to 2188 MHz a 1.8× increase, and to 2266 MHz a 2.3× improvement on average. A similar behavior is observed for the NX worker, where moving from the same initial CPU frequency to the next available frequency results in a 1.7×, 2×, and 2.3× higher QPS on average.

Regarding the number of available online CPUs, we observe a counterintuitive trend. For the AGX worker, increasing the number of online CPUs generally leads to a decrease in QPS on average. Specifically, utilizing 4 online CPUs results in an 8% decrease in QPS, while having 6 online CPUs leads to a 1.7× lower QPS compared to using 2 online CPUs. With 8 online CPUs, QPS remains nearly the same as with 2 CPUs, showing only a slight 1% drop on average. For the NX worker, QPS remains relatively stable. We observe a slight 2% decrease on average with 4 CPUs, followed by a 4% increase when using 6 CPUs compared to the 2-CPU configuration. To understand this behavior, we focus on the 2-CPU operating modes. On the AGX worker, the 2-CPU con-

figuration corresponds to Mode 4, which operates at a high frequency of 2100 MHz. In contrast, the 6-CPU configuration, where we observe the largest QPS drop, only includes Mode 2, which has a lower frequency of 1450 MHz. Even with 8 online CPUs, which include the best-performing Mode 6, the presence of Mode 1 operating at a lower frequency of 1200 MHz causes a decline in the QPS distribution. Similar observations hold for the NX worker. This suggests that operating frequency is a more critical factor than the number of online CPUs. If performance is the primary objective, it is often preferable to choose an operating mode with fewer CPUs but a higher frequency, rather than increasing the number of CPUs at the cost of reduced frequency.

As for the available power budget of the operating modes for the AGX worker, the MAXN power budget stands out with an average QPS of 18.4. The next best power budget is 15 Watts, which results in a 1.3× decrease, followed by 30 Watts, which leads to a 2× decrease compared to MAXN. Although one might expect that the 30-Watt mode would perform better than the 15-Watt mode, a deeper analysis of the operating modes reveals that the 15-Watt power budget is exclusively associated with Mode 6, which has the second-highest operating frequency. In contrast, the 30-Watt power budget includes multiple modes with significantly lower frequencies, leading to a lower average QPS. For the NX worker, the behavior follows expectations, with the lowest QPS values observed at 10 Watts, averaging 9.7 QPS. At 15 Watts and 20 Watts, the QPS values are similar, averaging 11.3. The similarity in performance between 15 Watts and 20 Watts is attributed to the wider range of operating frequencies available within these power budgets, preventing any single mode from standing out.

⋆ ***Key Outcome 5:*** *CPU frequency has the greatest impact on performance, outweighing the number of online CPUs, while power budget influences performance indirectly based on the frequency and modes it enables.*

Chapter 6

SynergAI Scheduling Framework

Based on the insights derived by the characterization and analysis presented in Section 2, we design `SynergAI`. Its main goal is to satisfy the QoS requirements of the deployed inference engines by reducing the number of QoS violations. `SynergAI` is illustrated in Figure 6.1 and aims to provide a synergetic Edge-to-Cloud solution and consists of two discrete phases, i.e. i) *Architecture-driven Performance Analysis & Characterization (Offline)* detailed in Sec. 1 and ii) *QoS-aware Run-time Scheduling & Deployment (Online)*, analyzed in Sec. 2.

1 Offline Phase: Architecture-driven Performance Analysis & Characterization

The offline phase aims to evaluate how each inference engine performs under various architecture-specific optimizations, driven by the characterization and analysis presented in Sec. 2. As input we provide the target inference engines intended for deployment within the proposed inference serving system, as well as the target Edge/Cloud nodes. More specifically, the inference engine consists of the backend (e.g., TensorFlow, ONNX Runtime), the pre-trained DNN model (e.g., ResNet), and the validation dataset (e.g., ImageNet), while the target nodes are characterized by their underlying architecture of workers (e.g., x86, ARM) and their operating modes. The result of the offline phase is the creation of a configuration dictionary for the discrete inference engines, which encapsulates the data to be used later for job scheduling during the Online phase (Sec. 2).

Initially, the *Performance-aware Configuration Generator* (1A) examines the available levels of parallelism for each architectural configuration, optimizing execution efficiency accordingly. This process involves configuring the degree of vertical scaling for each backend within the inference engine on each worker node, as highlighted in prior research in [4]. By systematically exploring parallelism options, the system ensures that inference workloads are optimally distributed across computational resources, maximizing throughput and min-

imizing latency. In addition to parallelism optimization, SynergAI integrates the *Architecture-aware Configuration Generator* 1B, designed to enhance efficiency in architectures where operating mode tuning is feasible. This module integrates the discrete operating modes available on specific hardware platforms, such as the AGX and NX boards, to assess their impact on performance within discrete power budgets. By dynamically selecting the most efficient operating mode configuration, SynergAI ensures that inference workloads are executed with an optimal balance between computational speed, threading, and efficiency, leading to architecture-tuned deployments. This dual optimization strategy—leveraging both parallelism exploration and architecture-aware tuning—enables SynergAI to adaptively configure heterogeneous Edge and Cloud nodes, optimizing resource utilization while meeting QoS constraints.

Once all potential configurations for an inference engine across the available workers are collected, a thorough *Design Space Exploration & Analysis* 1C is performed. This step involves systematically evaluating each configuration to identify the most efficient and high-performing settings for different workers, considering the specific operating modes. By carefully exploring these configurations, we gain valuable insights into the trade-offs between speed and efficiency, ultimately resulting in a well-balanced and optimized deployment strategy. During this exploration, we gather detailed performance metrics, such as the achieved QPS, total execution time, pre-processing time, actual computation time, threading, and more. These metrics provide a comprehensive view of the inference engine's performance under varying configurations. Following an in-depth analysis of the collected data, we identify the *Architecture-driven Optimal Deployments* 1D for each inference engine, selecting the best-performing configurations from all available architectures. This data is then stored in a structured database, ensuring easy access and retrieval when required. The final dataset forms the *Configuration Dictionary* 1E, which serves as a reference for the SynergAI orchestrator. This dictionary includes crucial elements for optimal inference serving, such as the model, backend, operating mode, and other system configurations. It acts as the foundation for job scheduling in the *Online Phase* (Sec. 2), enabling the orchestrator to make intelligent, architecture-tuned decisions that maximize the efficiency of the entire Edge-Cloud system.

2 Online Phase: QoS-aware Run-time Scheduling & Deployment

During the Online Phase, SynergAI continuously processes incoming inference workloads while ensuring compliance with the target QoS objectives. The scheduling policy constructs a *Job Queue* 2A, where each job represents an

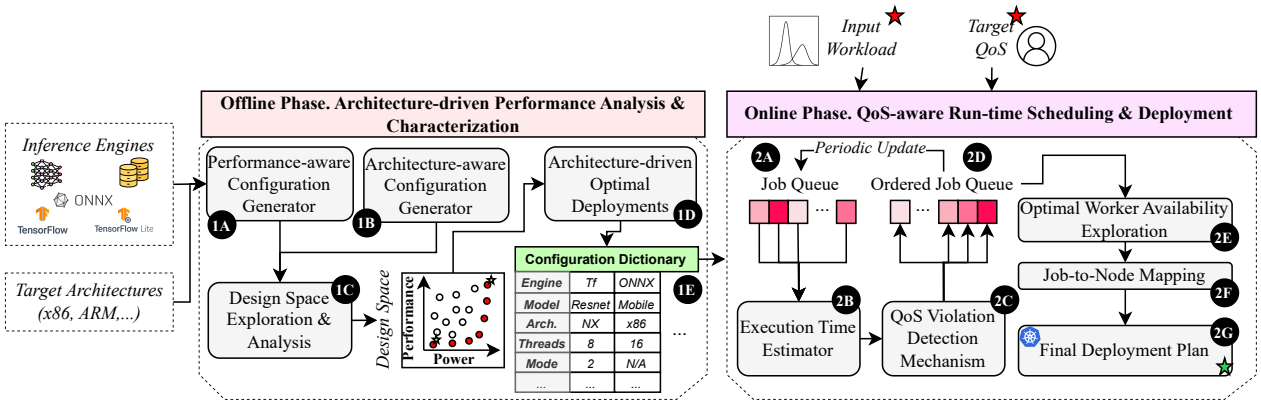


Figure 6.1: Overview of the Offline and Online Phases of the `SynergAI` Framework

inference engine with specific execution requirements, including the total number of queries to be processed and the target QoS. Let J be the set of incoming jobs, defined as $J = \{j_1, j_2, \ldots, j_N\}$, and W be the set of workers, defined as $W = \{w_1, w_2, \ldots, w_M\}$. The formulation of our target problem and decision-making process is presented in Eq. 6.1-Eq. 6.4, where each equation is evaluated for every job $j \in J$. The key parameters are briefly explained in Table 6.1.

$$T_{\text{Remaining},j} = T_{\text{QoS},j} - T_{\text{Waiting},j}, \quad \forall j \in J \tag{6.1}$$

$$T_{\text{Estimated},j,w} = T_{\text{Pre-processing},j} + \frac{q}{QPS^{c^*_{j,w}}}, \quad \forall j \in J, \forall w \in W \tag{6.2}$$

$$W_{\text{acceptable},j} = \{w \in W \mid T_{\text{Remaining},j} \geq T_{\text{Estimated},j,w}\}, \quad \forall j \in J \tag{6.3}$$

$$w^*_j = \arg \min_{w \in W_{\text{acceptable},j}} T_{\text{Estimated},j,w} \tag{6.4}$$

Table 6.1: Key System and Scheduler Parameters

Denotation	Short Description
q	Number of queries to be executed for the given inference
J	Set of inference jobs in the system, where each job $j \in J$.
W	Set of worker nodes in the system, where each worker $w \in W$.
$c^*_{j,w}$	Optimal configuration for job j on worker w, maximizing QPS.
w^*_j	The optimal worker for job j that minimizes execution time while meeting QoS constraints.
$T_{\text{QoS},j}$	Time specified by the user for executing job j.
$T_{\text{Waiting},j}$	Time elapsed since job j was submitted to the queue.
$T_{\text{Remaining},j}$	Time remaining before a QoS violation occurs for job j, calculated as $T_{\text{QoS},j} - T_{\text{Waiting},j}$.
$T_{\text{Pre-processing},j}$	Pre-processing time for job j, derived from profiling.
$T_{\text{Estimated},j,w}$	Estimated execution time for job j on worker w, including pre-processing time and execution time per query.
$QPS^{c^*_{j,w}}$	Queries per Second (QPS) achieved by the optimal configuration $c^*_{j,w}$ for job j on worker w.
$W_{\text{acceptable},j}$	Set of workers capable of completing job j within the remaining allowed time.

Let $T_{\text{QoS},j}$ denote the time specified by the user for executing job j, and let $T_{\text{Waiting},j}$ represent the time elapsed since the job j was submitted to the queue. We define the remaining time before a QoS violation for j as $T_{\text{Remaining},j}$, expressed in Eq. 6.1. As $T_{\text{Waiting},j}$ increases, $T_{\text{Remaining},j}$ decreases, approaching zero, indicating an increasing urgency for execution. To effectively manage job prioritization and prevent QoS violations, `SynergAI` continuously monitors these time constraints. Subsequently, the queued jobs are forwarded to the *Execution Time Estimator* 2B, which is the key enabler of adaptive scheduling, allowing `SynergAI` to dynamically determine job execution order. Utilizing the *Configuration Dictionary* 1D, `SynergAI` selects the optimal configuration for each worker that maximizes QPS for a given inference engine, denoted as $c^*_{j,w}$ for each job j and worker w. Given a request to execute q queries and the profiled pre-processing time for job j, denoted as $T_{\text{Pre-processing},j}$, the estimated execution time $T_{\text{Estimated},j,w}$ is formulated in Eq. 6.2, where $QPS^{c^*_{j,w}}$ represents the QPS achieved by configuration $c^*_{j,w}$. This information is incorporated into the *QoS Violation Detection Mechanism* 2C, whose goal is to identify jobs at risk of exceeding their QoS constraints, determine the optimal worker for deployment, and prioritize jobs with a higher probability of violation. The system evaluates the urgency of each job by computing the difference between $T_{\text{Remaining},j}$ and $T_{\text{Estimated},j,w}$ for each worker w. As the difference approaches zero, urgency increases, while a negative value indicates an inevitable QoS violation.

The final worker selection is determined through a set of acceptable workers, which can guarantee QoS compliance given the remaining time $T_{\text{Remaining},j}$, as defined in Eq. 6.3. These workers are then sorted in ascending order by their estimated execution time. The optimal worker w^*_j for the given job is the worker that minimizes the estimated execution time $T_{\text{Estimated},j,w}$, as formulated in Eq. 6.4. This guarantees that jobs are assigned to the fastest available worker nodes, reducing execution time while meeting QoS requirements. Additionally, by maintaining a sorted list of acceptable nodes for each job, we prevent increased waiting times by checking the next best available node when the first choice is occupied. Once urgency values for all jobs are computed and the optimal assignments are identified, the queue is sorted in descending order based on urgency, ensuring that jobs at the highest risk of QoS violations are prioritized, composing the *Ordered Job Queue* 2D. If no worker can complete a job within its required execution time (i.e., a QoS violation occurs), the job is de-prioritized and moved to the end of the queue, allowing higher-priority jobs with active QoS constraints to be processed first. To ensure continuous optimization, a periodic update mechanism reassesses all jobs in the queue, mapping them to the most suitable worker based on updated waiting times and dynamically reordering the queue as needed. Jobs are dequeued sequen-

tially, each associated with a list of $(w, c^*_{j,w})$ pairs, where each pair consists of a worker and its optimal configuration for the given inference engine. Finally, *Optimal Worker Availability Exploration* (2E) iterates through the sorted set $W_{\text{acceptable},j}$, checking each worker's availability until it finds the first available worker: $(w_j,\, c^*_{j,w^*_j})$, starting from the optimal worker w^*_j and then performs the final *Job-to-Node Mapping* (2F). Once the job is mapped to the corresponding node, the *Final Deployment Plan* (2G) is derived accordingly and deployed via Kubernetes. This iterative process continues until all jobs are scheduled and executed successfully within the Edge/Cloud cluster.

Chapter 7

Experimental Evaluation

In this section, we present the experimental evaluation of SynergAI. First, we explain the experimental setup and the mechanisms used for comparison. Finally, we present and analyze the results of our experiments and the impact of SynergAI.

1 Experiment Overview and Baselines Description

We assess the SynergAI scheduler through multiple experiments to evaluate its efficiency. Each experiment consists of a set of 24 inference engines to be served. Each inference engine specifies a required number of queries to be executed along with its QoS constraints, meaning it expects to complete these queries within a specified time limit. To determine the demands, we aggregated execution times across all configurations and workers for each inference engine, considering the varying query loads of 3900, 1950, 975, and 488 queries per experimental run, as shown in Figure 7.1. From this distribution, we extracted the median values to define a demand-low-intensity (DL) demand set and the 25%-ile values to represent a demand-high-intensity workload (DH). These values are summarized in Table 7.1. Regarding the arrival of inference requests at SynergAI's scheduler, we follow a Poisson distribution, as done in prior research [1], since it effectively models request arrivals by capturing independent events, following an exponential inter-arrival time, reflecting real-world variability, and offering mathematical simplicity for analysis. The λ parameter of the distribution is derived from the data gathered during characterization. Specifically, Figure 7.2 illustrates how we aggregate execution times for all inference engines across all configurations and workers, extracting the median and 25%-ile values from the distribution to define a low request frequency (FL) and a high request frequency (FH). In total, we create four distinct experiments: DL-FL, DL-FH, DH-FL, and DH-FH, each with increasing difficulty to evaluate our scheduling system.

We compare the SynergAI scheduler with various other scheduling methods, ranging from standard approaches to SotA scheduling systems. Besides

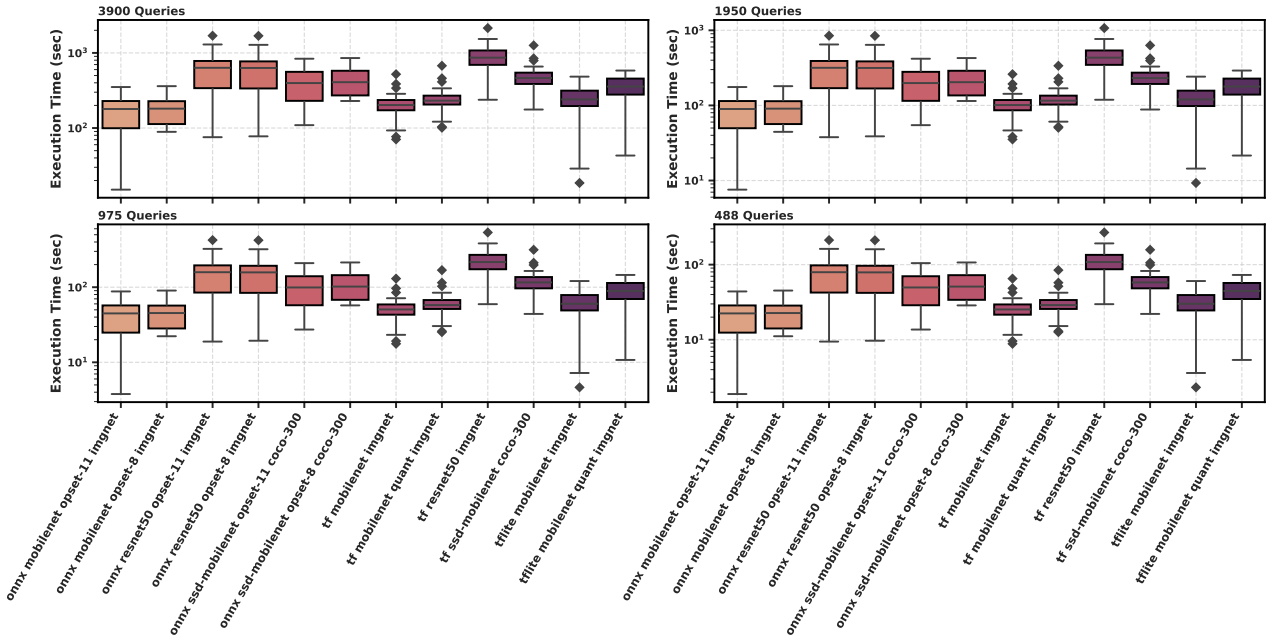


Figure 7.1: Aggregated Execution Times Across All Configurations and Workers for Each Inference Engine

our scheduling approach, we implement five additional scheduling systems: i) *Round Robin* (`RR`), which allocates inference engines to workers in a circular sequence, regardless of worker capabilities or job demands, ensuring a fair distribution of workload across the system; ii) *Strict Round Robin* (`SRR`), a variation of `RR`, where each job is strictly assigned to the next worker and waits for its availability, aiming for perfect distribution but risking increased waiting times and demand violations; iii) *Least Recently Used* (`LRU`), which assigns jobs to the worker that has been idle the longest to prevent starvation and promote an even workload distribution; iv) *Most Recently Used* (`MRU`), which allocates jobs to the most recently active node that is available, although it risks node starvation, as some nodes may remain underutilized; v) *Best Effort* (`BE`), which follows a greedy policy, iterating from the strongest worker to the weakest until

Table 7.1: Inference Engine Performance Metrics Across Query Sets

Inference Engine	3900 Queries		1950 Queries		975 Queries		488 Queries	
	25%-ile	Median	25%-ile	Median	25%-ile	Median	25%-ile	Median
ONNX MobileNet opset-11	99.40	179.23	49.70	89.61	24.85	44.81	12.44	22.43
ONNX MobileNet opset-8	112.92	181.18	56.46	90.59	28.23	45.30	14.13	22.67
ONNX ResNet50 opset-11	339.48	633.63	169.74	316.82	84.87	158.41	42.48	79.29
ONNX ResNet50 opset-8	336.46	630.56	168.23	315.28	84.12	157.64	42.10	78.90
ONNX SSD-MobileNet opset-11	230.09	396.95	115.04	198.47	57.52	99.24	28.79	49.67
ONNX SSD-MobileNet opset-8	271.74	407.74	135.87	203.87	67.94	101.93	34.00	51.02
TF MobileNet	172.31	202.33	86.15	101.16	43.08	50.58	21.56	25.32
TF MobileNet Quant	205.44	231.36	102.72	115.68	51.36	57.84	25.71	28.95
TF ResNet50	693.64	865.31	346.82	432.65	173.41	216.33	86.79	108.27
TF SSD-MobileNet	386.29	463.02	193.14	231.51	96.57	115.76	48.34	57.94
TFLite MobileNet	196.18	240.45	98.09	120.22	49.05	60.11	24.55	30.09
TFLite MobileNet Quant	278.65	356.73	139.33	178.37	69.66	89.18	34.87	44.64

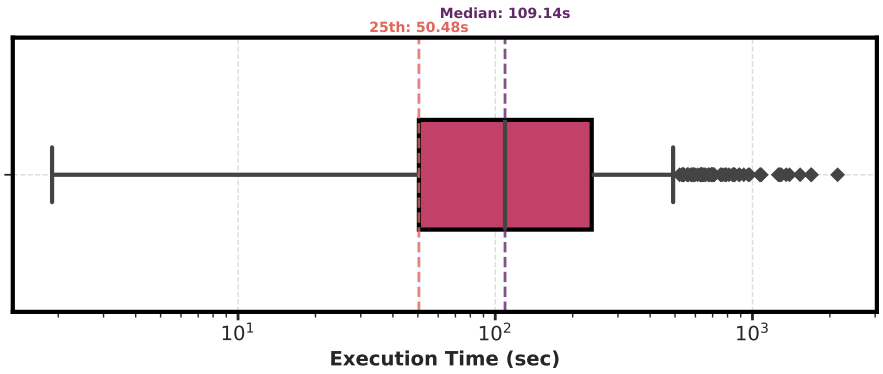


Figure 7.2: Aggregated Execution Time for All Queries and Models Across All Configurations, Workers and Inference Engines

an available is found, and then assigns the job to that worker for execution. Additionally, we implement from scratch and compare against a SotA scheduling solution derived by [1]. We focus on the proposed solution without model slicing, namely SLO Minimum-Average-Expected-Latency (`SLO-MAEL`), since our work does not focus on the model slicing techniques. `SLO-MAEL` aims to reduce QoS violations and the decision-making relies on evaluating all possible mappings of inference engines to workers using a scoring system. The performance of each scheduling system is evaluated using the following metrics: a) total number of violations, b) waiting time, which refers to the duration the inference engine spends in the queue, c) end-to-end execution time, which includes both waiting and execution times, and d) average excess time, which measures the average time a job exceeds its desired execution time. This is calculated as the difference between the actual and desired times, with excess time clipped to zero for jobs that do not violate the QoS.

2 Detailed Analysis of `SynergAI` versus Defined Baselines

Figure 7.3 presents the results of the `DL-FL` experiment. Analyzing the number of violations, the `SRR` scheduler records the highest count at 18, followed by `LRU` with 14 and `RR` with 11. The `MRU` scheduler incurs 5 violations, while `BE` reports 3. `SynergAI` achieves the fewest violations, with only 2 recorded. Beyond violation counts, `SynergAI` also demonstrates superior performance in end-to-end execution time. Its execution time distribution ranges from a minimum of 7.4 seconds to a maximum of 4.3 minutes, with an average of 1.24 minutes and a 99%-ile of 4.3 minutes. The 99%-ile metric is particularly important, as it captures the worst-case execution time for the slowest 1% of jobs, ensuring system reliability under peak loads. `SynergAI` consistently outperforms all other schedulers, with its 99%-ile execution time being lower by factors of 2.1×, 2.3×, 2.4×, 2.4×, and 3.6× compared to `BE`, `MRU`, `RR`, `LRU`, and `SRR`, respectively. The

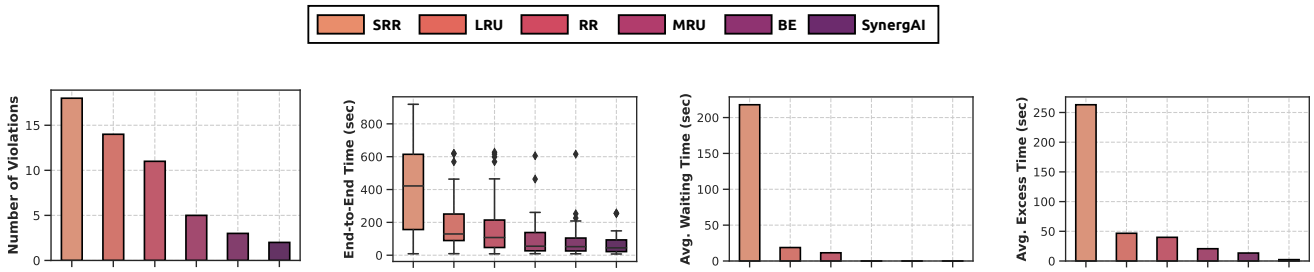


Figure 7.3: Comparison of SynergAI with other Scheduling Systems in the DL/FL Experiment

average waiting time provides further insights into scheduling efficiency. As expected, the SRR scheduler exhibits the longest waiting time at 3.6 minutes due to its strict job rotation policy, which evenly distributes workloads but increases delays and violation risks. The LRU and RR schedulers show shorter waiting times of 18.8 seconds and 11.5 seconds, respectively. Notably, both MRU and BE, along with SynergAI, experience no waiting time. SynergAI's advantage comes from its ability to identify the optimal configuration for each inference engine across worker nodes. This underscores the importance of the Offline Phase, as other schedulers rely on predefined configurations, typically selecting the worker with the highest CPU resources. By leveraging its learned knowledge, SynergAI surpasses these conventional approaches. Moreover, due to its adaptive configuration strategy, even for the two jobs that exceeded their QoS targets, the average excess time remains exceptionally low at just 2.3 seconds. This is significantly lower compared to the excess times recorded by other schedulers: 13.4 seconds for BE, 20.7 seconds for MRU, 39.9 seconds for RR, 46.7 seconds for LRU, and 4.3 minutes for SRR.

In the DL-FH experiment, we increase the arrival rate of inference engines to evaluate how the scheduling schemes perform under higher workload conditions. Figure 7.4 presents the results of this experiment. Among the schedulers, SRR exhibits the highest number of violations, with 21 violations, meaning only three jobs met their constraints. The LRU scheduler follows with 15 violations, while RR, MRU, and BE each record 16 violations, indicating a performance decline under heavier load. In contrast, SynergAI achieves the best performance, recording the fewest violations at just 6. For the execution time distribution, SynergAI performs remarkably well, with a range from a minimum of 9.5 seconds to a maximum of 12 minutes, an average of 2.3 minutes, and a 99%-ile of 10.4 minutes. Regarding waiting time, SRR has the highest at 13.3 minutes, while LRU and RR exhibit shorter waiting times of 1.36 minutes and 1.56 minutes, respectively. Both MRU and BE display waiting times of 1.41 minutes and 1.42 minutes, explaining the higher number of violations for these schedulers. On the other hand, SynergAI has the smallest average waiting time at just 49.2 seconds. Finally, regarding excess time, SynergAI maintains an average excess

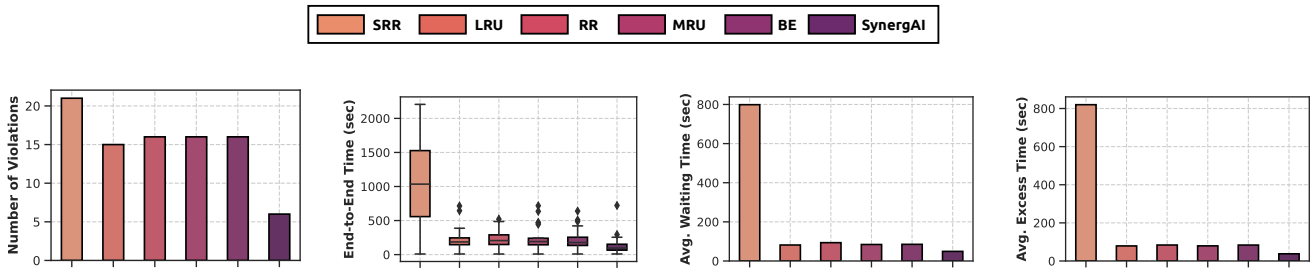


Figure 7.4: Comparison of `SynergAI` with other Scheduling Systems in the `DL/FH` Experiment

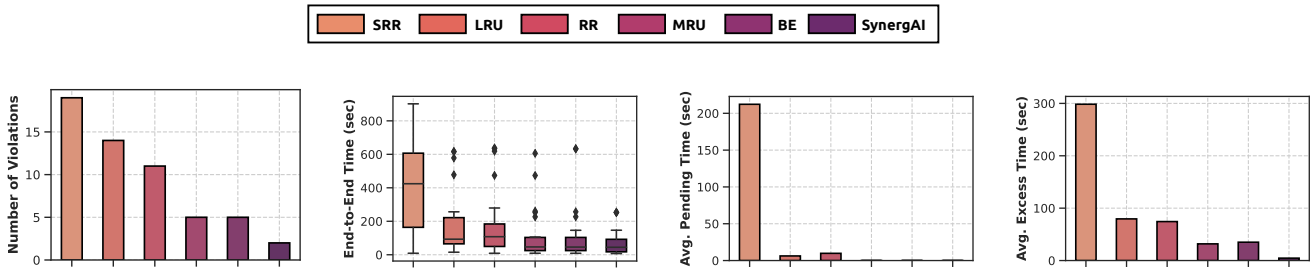


Figure 7.5: Comparison of `SynergAI` with other Scheduling Systems in the `DH/FL` Experiment

time of 37.7 seconds for the jobs that led to violations. This is significantly lower compared to the excess times recorded by other schedulers: 1.4 minutes for `BE`, 1.3 minutes for `MRU`, 1.4 minutes for `RR`, 1.3 minutes for `LRU`, and 13.7 minutes for `SRR`.

We also evaluate the scheduler's performance under high demands but lower frequency of arrivals in the `DH-FL` experiment. The results of this experiment are presented in Figure 7.5. In terms of violations, the `SRR` scheduler records the highest number at 19, followed by `LRU` with 14, and `RR` with 11 violations. Both `BE` and `MRU` show better performance with 5 violations each, while `SynergAI` achieves a superior outcome with just 2 violations, the fewest among all schedulers. For execution time distribution, `SynergAI` delivers exceptional results, with times ranging from a minimum of 7.24 seconds to a maximum of 4.24 minutes, an average of 1.20 minutes, and a 99%-ile of 4.23 minutes. When comparing the 99%-ile metric, `SynergAI` outperforms all other schedulers by significant margins: 2.50× lower than `BE`, 2.43× lower than `LRU`, 2.27× lower than `MRU`, 2.51× lower than `RR`, and 3.51× lower than `SRR`. The analysis of waiting times reveals that `SRR` experiences the longest average waiting time at 3.54 minutes, while `RR` and `LRU` show moderate waiting times of 9.80 seconds and 6.26 seconds, respectively. Notably, `SynergAI`, `BE`, and `MRU` all exhibit no waiting times, similarly to the `DL-FL` experiment. `SynergAI`'s efficiency is further reflected in the average excess time metric, where it maintains an impressively low average of just 4.59 seconds for jobs that exceeded their QoS targets. This is significantly better than other schedulers: 31.84 seconds for `MRU`, 35.03 seconds for `BE`, 1.24 minutes for `RR`, 1.33 minutes for `LRU`, and 4.97 minutes for `SRR`.

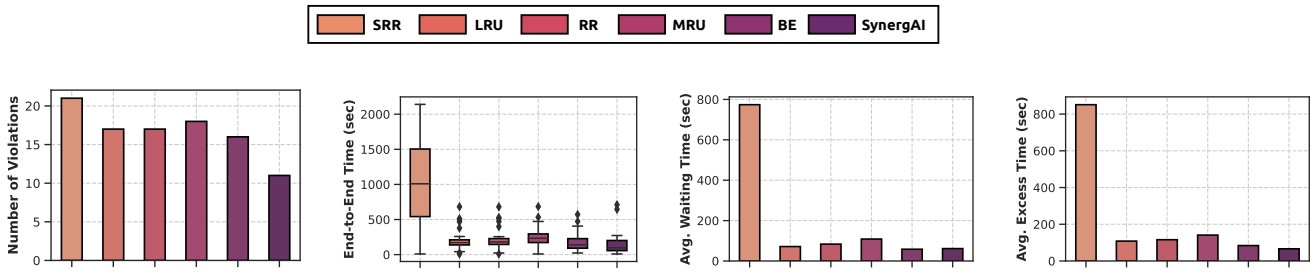


Figure 7.6: Comparison of SynergAI with other Scheduling Systems in the DH/FH Experiment

Finally, in the DH-FH experiment, we not only increase the arrival rate of inference engines but also raise the QoS demands for each engine, making this the most challenging experiment in terms of load for all the schedulers. Figure 7.6 presents the results of this experiment. Once again, SRR records the highest number of violations, totaling 21. The other schedulers show a slight increase in violations compared to the previous experiment, with LRU, RR, MRU, and BE recording 17, 17, 18, and 16 violations, respectively. In contrast, SynergAI delivers the best performance even in this demanding scenario, achieving the fewest violations with only 11. The execution time distribution of SynergAI ranges from a minimum of 9.3 seconds to a maximum of 11.8 minutes, with an average of 2.75 minutes and a 99%-ile of 11.6 minutes. When comparing the 99%-ile, SynergAI outperforms the other schedulers, with a value 1.3× lower on average, showcasing its superior efficiency. As for waiting times, SynergAI has the lowest average waiting time of approximately 1 minute. The next slowest schedulers are BE at 1.1 minutes, followed by LRU at 1.2 minutes, RR at 1.4 minutes, MRU at 1.8 minutes, and SRR at 12.9 minutes. Regarding excess time, SynergAI maintains an average excess time of 1.1 minutes for the jobs that resulted in violations, which is significantly lower than the excess times recorded by the other schedulers: 1.4 minutes for BE, 2.35 minutes for MRU, 1.9 minutes for RR, 1.8 minutes for LRU, and 14.2 minutes for SRR. To summarize all the experiments, SynergAI achieves an average reduction of 7.1× in QoS violations and 5.3× in excess time, respectively.

3 Comparison with SotA Scheduling System

We also compare SynergAI with the SotA scheduling system proposed by [1] (SLO-MAEL). We once again compare SynergAI with SLO-MAEL across the four distinct experiments we previously analyzed: DL-FL, DL-FH, DH-FL, and DH-FH, each representing scenarios with escalating difficulty and load. For this evaluation, we utilize the same metrics as in our previous analysis to ensure consistency. Figure 7.7 presents the results from all conducted experiments. In the DL-FL experiment, SLO-MAEL results in 5 violations. The end-to-end execution time varies significantly, ranging from a minimum of 10.7 seconds to a maximum

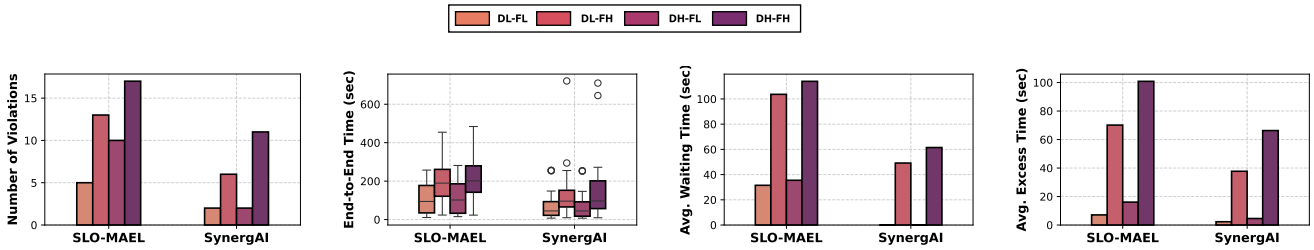


Figure 7.7: Comparison of SynergAI with SLO Minimum-Average-Expected-Latency (SLO-MAEL) [1] for all Experiments

of 4.4 minutes, with an average of 1.8 minutes and a 99%-ile of 4.2 minutes. In contrast, as previously shown, SynergAI leads to only 2 violations while achieving a 1.5× reduction in average execution time. This performance gap can be primarily attributed to the waiting time experienced by tasks. SLO-MAEL exhibits an average waiting time of 31.5 seconds, whereas SynergAI completely eliminates pending time, allowing for more efficient scheduling and execution. Although SLO-MAEL utilizes a scoring system to optimize task-to-worker placement based on current queue conditions, it lacks adaptive rescheduling capabilities and fails to account for the optimal resource configurations of each worker. Consequently, this leads to a higher number of violations and longer waiting times. The drawbacks of SLO-MAEL are also evident in the average excess execution time, which measures how much longer tasks take beyond their expected duration. Here, SLO-MAEL records an average excess time of 7.1 seconds, which is 3.1× higher than that of our approach. This further highlights the inefficiencies in its scheduling decisions compared to SynergAI, which optimally balances task assignments to minimize delays and violations.

For the DL-FH, DH-FL, and DH-FH experiments, which present increasingly complex and challenging scenarios, we observe a consistent pattern. SLO-MAEL results in 13 violations for DL-FH, 10 for DH-FL, and 17 for DH-FH, which are 2.2×, 5×, and 1.6× higher, respectively, compared to the number of violations seen with SynergAI. Execution times are also noticeably higher for SLO-MAEL, with average times that are 1.5× longer for DL-FH, 1.56× longer for DH-FL, and 1.3× longer for DH-FH. Similarly, waiting times for SLO-MAEL are significantly greater than those observed with SynergAI, standing at 1.9× higher in DL-FH, while in DH-FL, the difference is even more pronounced with SLO-MAEL's average waiting time of 35.5 seconds compared to SynergAI's complete elimination of delays. For DH-FH, SLO-MAEL's waiting time is 1.5× higher than SynergAI. Furthermore, the average excess execution time further emphasizes the inefficiencies of SLO-MAEL, showing 2.2× higher values in DL-FH, 3.5× higher in DH-FL, and 1.85× higher in DH-FH compared to SynergAI. These results clearly demonstrate that SynergAI outperforms SLO-MAEL in terms of scheduling efficiency by reducing violations, execution times, waiting times, and excess execution,

positioning it as a more effective solution for inference scheduling.

Chapter 8

Conclusion and Future Work

1 Summary

In this work, we introduce `SynergAI`, an intelligent scheduling framework designed to dynamically optimize workload distribution across heterogeneous edge and cloud environments. By leveraging extensive performance characterization, `SynergAI` efficiently allocates inference-serving workloads to minimize QoS violations while maximizing resource utilization. Our framework accounts for the trade-offs between performance and architecture-operating modes, ensuring a well-balanced deployment strategy across the computing continuum. Through seamless integration within a Kubernetes-based ecosystem, `SynergAI` demonstrates its effectiveness in handling diverse inference-serving scenarios, adapting to varying workloads, and improving overall system efficiency. Our findings show that architecture-driven inference serving facilitates optimized, efficient deployments on emerging hardware platforms, resulting in an average reduction of 2.4× in QoS violations compared to a SotA solution.

2 Future Work

For future work, we plan to extend `SynergAI` to incorporate GPU acceleration across the edge-cloud continuum, which would significantly enhance performance for computation-intensive DNN inference tasks. Additionally, we aim to integrate DNN partitioning capabilities into our framework, enabling the system to automatically split and distribute neural network layers across edge devices based on their computational capabilities and current load. Such an approach would further reduce latency and resource consumption by optimizing both where and how inference tasks are executed. By combining GPU acceleration with intelligent DNN partitioning, SynergAI could achieve even greater reductions in QoS violations while maintaining energy efficiency across heterogeneous edge-cloud environments.

Bibliography

[1] W. Seo, S. Cha, Y. Kim, J. Huh, and J. Park, "Slo-aware inference scheduler for heterogeneous processors in edge platforms," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 18, no. 4, pp. 1–26, 2021.

[2] L. Floridi and M. Chiriatti, "Gpt-3: Its nature, scope, limits, and consequences," *Minds and Machines*, vol. 30, pp. 681–694, 2020.

[3] A. Liu, B. Feng, B. Wang, B. Wang, B. Liu, C. Zhao, C. Dengr, C. Ruan, D. Dai, D. Guo *et al.*, "Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model," *arXiv preprint arXiv:2405.04434*, 2024.

[4] A. Ferikoglou, P. Chrysomeris, A. Tzenetopoulos, M. Katsaragakis, D. Masouros, and D. Soudris, "Iris: interference and resource aware predictive orchestration for ml inference serving," in *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*. IEEE, 2023, pp. 1–12.

[5] J. Ding, R. Cao, I. Saravanan, N. Morris, and C. Stewart, "Characterizing service level objectives for cloud services: Realities and myths," in *2019 IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, 2019, pp. 200–206.

[6] Y. E. Wang, G.-Y. Wei, and D. Brooks, "Benchmarking tpu, gpu, and cpu platforms for deep learning," *arXiv preprint arXiv:1907.10701*, 2019.

[7] "Vertex AI," https://cloud.google.com/vertex-ai, accessed: 24-03-2025.

[8] "Amazon SageMaker," https://aws.amazon.com/sagemaker/, accessed: 24-03-2025.

[9] "Azure Machine Learning," https://azure.microsoft.com/en-us/products/machine-learning, accessed: 24-03-2025.

[10] R. High, "The era of cognitive systems: An inside look at ibm watson and how it works," *IBM Corporation, Redbooks*, vol. 1, p. 16, 2012.

[11] K. B. Letaief, Y. Shi, J. Lu, and J. Lu, "Edge artificial intelligence for 6g: Vision, enabling technologies, and applications," *IEEE journal on selected areas in communications*, vol. 40, no. 1, pp. 5–36, 2021.

[12] A. K. Kakolyris, M. Katsaragakis, D. Masouros, and D. Soudris, "Roadrunner: Collaborative dnn partitioning and offloading on heterogeneous edge systems," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2023, pp. 1–6.

[13] X. Guo, F. Dong, D. Shen, Z. Huang, and J. Zhang, "Resource-efficient dnn inference with early exiting in serverless edge computing," *IEEE Transactions on Mobile Computing*, 2024.

[14] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 77–88, 2013.

[15] Y. He, J. Fang, F. R. Yu, and V. C. Leung, "Large language models (llms) inference offloading and resource allocation in cloud-edge computing: An active inference approach," *IEEE Transactions on Mobile Computing*, 2024.

[16] N. Wang, R. Zhou, L. Jiao, R. Zhang, B. Li, and Z. Li, "Preemptive scheduling for distributed machine learning jobs in edge-cloud networks," *IEEE Journal on Selected Areas in Communications*, vol. 40, no. 8, pp. 2411–2425, 2022.

[17] Kubernetes Project, "Kubernetes: Production-grade container orchestration," https://kubernetes.io, 2024, accessed: 2025-03-28.

[18] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou *et al.*, "Mlperf inference benchmark," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 446–459.

[19] "Tensorflow Serving," https://www.tensorflow.org/tfx/guide/serving, accessed: 25-03-2025.

[20] "Tensorflow Lite," https://github.com/tensorflow/tflite-micro, accessed: 25-03-2025.

[21] "ONNX Runtime," https://onnxruntime.ai/, accessed: 25-03-2025.

[22] E. Limonova, D. Alfonso, D. Nikolaev, and V. V. Arlazarov, "Resnet-like architecture with low hardware requirements," in *2020 25th International Conference on Pattern Recognition (ICPR)*. IEEE, 2021, pp. 6204–6211.

[23] I. Cohen, Y. Huang, J. Chen, J. Benesty, J. Benesty, J. Chen, Y. Huang, and I. Cohen, "Pearson correlation coefficient," *Noise reduction in speech processing*, pp. 1–4, 2009.

[24] M. Van Baalen, A. Kuzmin, S. S. Nair, Y. Ren, E. Mahurin, C. Patel, S. Subramanian, S. Lee, M. Nagel, J. Soriaga *et al.*, "Fp8 versus int8 for efficient deep learning inference," *arXiv preprint arXiv:2303.17951*, 2023.

[25] PyTorch, "Int8 quantization for x86 cpu in pytorch," https://pytorch.org/blog/int8-quantization/?utm$_source$ = *chatgpt.com*, 2024, *accessed* : 2025 − 03 − 28.

[26] S. Pouyanfar, S. Sadiq, Y. Yan, H. Tian, Y. Tao, M. P. Reyes, M.-L. Shyu, S.-C. Chen, and S. S. Iyengar, "A survey on deep learning: Algorithms, techniques, and applications," *ACM Computing Surveys (CSUR)*, vol. 51, no. 5, pp. 1–36, 2018.

[27] Y. Yao, J. Duan, K. Xu, Y. Cai, Z. Sun, and Y. Zhang, "A survey on large language model (llm) security and privacy: The good, the bad, and the ugly," *High-Confidence Computing*, p. 100211, 2024.

[28] Y. Alali, F. Harrou, and Y. Sun, "A proficient approach to forecast covid-19 spread via optimized dynamic machine learning models," *Scientific Reports*, vol. 12, no. 1, p. 2467, 2022.

[29] M. Y. Shaheen, "Applications of artificial intelligence (ai) in healthcare: A review," *ScienceOpen Preprints*, 2021.

[30] A. A. Khan, A. A. Laghari, and S. A. Awan, "Machine learning in computer vision: A review." *EAI Endorsed Transactions on Scalable Information Systems*, vol. 8, no. 32, 2021.

[31] P. Gogas and T. Papadimitriou, "Machine learning in economics and finance," *Computational Economics*, vol. 57, pp. 1–4, 2021.

[32] "NVidia AI Enterprise," https://www.nvidia.com/en-eu/data-center/products/ai-enterprise/, accessed: 24-03-2025.

[33] J. Grohmann, P. K. Nicholson, J. O. Iglesias, S. Kounev, and D. Lugones, "Monitorless: Predicting performance degradation in cloud applications with machine learning," in *Proceedings of the 20th international middleware conference*, 2019, pp. 149–162.

[34] P. Yang, N. Xiong, and J. Ren, "Data security and privacy protection for cloud storage: A survey," *Ieee Access*, vol. 8, pp. 131 723–131 740, 2020.

[35] R. Singh and S. S. Gill, "Edge ai: a survey," *Internet of Things and Cyber-Physical Systems*, vol. 3, pp. 71–92, 2023.

[36] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner, "Ai and ml accelerator survey and trends," in *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2022, pp. 1–10.

[37] S. Imambi, K. B. Prakash, and G. Kanagachidambaresan, "Pytorch," *Programming with TensorFlow: solution for edge computing applications*, pp. 87–104, 2021.

[38] E. Li, L. Zeng, Z. Zhou, and X. Chen, "Edge ai: On-demand accelerating deep neural network inference via edge computing," *IEEE transactions on wireless communications*, vol. 19, no. 1, pp. 447–457, 2019.

[39] L. Shi, Z. Xu, Y. Sun, Y. Shi, Y. Fan, and X. Ding, "A dnn inference acceleration algorithm combining model partition and task allocation in heterogeneous edge computing system," *Peer-to-Peer Networking and Applications*, vol. 14, no. 6, pp. 4031–4045, 2021.

[40] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and qos-aware cluster management," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 127–144, 2014.

[41] M. Xue, H. Wu, R. Li, M. Xu, and P. Jiao, "Eosdnn: An efficient offloading scheme for dnn inference acceleration in local-edge-cloud collaborative environments," *IEEE Transactions on Green Communications and Networking*, vol. 6, no. 1, pp. 248–264, 2021.

[42] M. Xue, H. Wu, G. Peng, and K. Wolter, "Ddpqn: An efficient dnn offloading strategy in local-edge-cloud collaborative environments," *IEEE Transactions on Services Computing*, vol. 15, no. 2, pp. 640–655, 2021.

[43] S. Shahhosseini, D. Seo, A. Kanduri, T. Hu, S.-S. Lim, B. Donyanavard, A. M. Rahmani, and N. Dutt, "Online learning for orchestration of inference in multi-user end-edge-cloud networks," *ACM Transactions on Embedded Computing Systems*, vol. 21, no. 6, pp. 1–25, 2022.

[44] "Triton Inference Server," https://developer.nvidia.com/nvidia-triton-inference-server, accessed: 25-03-2025.

[45] Y. Xiao, L. Xiao, K. Wan, H. Yang, Y. Zhang, Y. Wu, and Y. Zhang, "Reinforcement learning based energy-efficient collaborative inference for mobile edge computing," *IEEE Transactions on Communications*, vol. 71, no. 2, pp. 864–876, 2022.

[46] A. Archet, N. Ventroux, N. Gac, and F. Orieux, "Energy-efficient use of an embedded heterogeneous soc for the inference of cnns," in *2023 26th Euromicro Conference on Digital System Design (DSD)*. IEEE, 2023, pp. 30–38.

[47] A. Tzenetopoulos, D. Masouros, S. Xydis, and D. Soudris, "Interference-aware workload placement for improving latency distribution of converged hpc/big data cloud infrastructures," in *Embedded Computer Systems: Architectures, Modeling, and Simulation: 21st International Conference, SAMOS 2021, Virtual Event, July 4–8, 2021, Proceedings*. Springer, 2022, pp. 108–123.

[48] T. Patel and D. Tiwari, "Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 193–206.

[49] P. Garefalakis, K. Karanasos, P. Pietzuch, A. Suresh, and S. Rao, "M edea: scheduling of long running applications in shared production clusters," in *Proceedings of the Thirteenth EuroSys Conference*. ACM, 2018, p. 4.

[50] M. Li, Y. Li, Y. Tian, L. Jiang, and Q. Xu, "Appealnet: An efficient and highly-accurate edge/cloud collaborative architecture for dnn inference," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 409–414.

[51] K. Rao, G. Coviello, P. Benedetti, C. Giuseppe De Vita, G. Mellone, and S. Chakradhar, "Eco-llm: Llm-based edge cloud optimization," in *Proceedings of the 2024 Workshop on AI For Systems*, 2024, pp. 7–12.

[52] D. R. Tripathi, H. T. Deshlahare, and R. R. Markhande, "Edge-cloud continuum: Integrating edge computing and cloud computing for iot applications," *International Journal for Research in Applied Science & Engineering Technology (IJRASET)*, vol. 12, no. 10, pp. 335–342, 2024.

[53] K. Cao, Y. Liu, G. Meng, and Q. Sun, "An overview on edge computing research," *IEEE Access*, vol. 8, pp. 85 714–85 728, 2020.

[54] D. Milojicic, "The edge-to-cloud continuum," *Computer*, vol. 53, no. 11, pp. 16–25, 2020.

[55] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[56] W. Z. Khan, E. Ahmed, S. Hakak, I. Yaqoob, and A. Ahmed, "Edge computing: A survey," *Future Generation Computer Systems*, vol. 97, pp. 219–235, 2019.