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

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Deep Reinforcement Learning for Tail-Latency Regulation in Co-located Applications through Cooperative Core and Cache Allocation

Κιμωνίδης Αλέξανδρος
Α.Μ. : 03116761

Επιβλέπων : Δημήτριος Ι. Σούντρης
Καθηγητής ΕΜΠ

Αθήνα
Νοέμβριος 2021

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

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Deep Reinforcement Learning for Tail-Latency Regulation in Co-located Applications through Cooperative Core and Cache Allocation

Κιμωνίδης Αλέξανδρος
Α.Μ. : 03116761

Επιβλέπων : Δημήτριος Ι. Σούντρης
Καθηγητής ΕΜΠ

Τριμελής Επιτροπή Εξέτασης

(Υπογραφή)	(Υπογραφή)	(Υπογραφή)
.....	.....	.....
Δημήτριος Σούντρης Καθηγητής ΕΜΠ	Παναγιώτης Τσανάκας Καθηγητής ΕΜΠ	Γεώργιος Γκούμας Επίκουρος Καθηγητής ΕΜΠ

Ημερομηνία Εξέτασης:
18 Νοεμβρίου 2021

Copyright ©- All rights reserved Κιμωνίδης Αλέξανδρος, 2021.
Με επιφύλαξη κάθε δικαιώματος.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ΄ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

(Υπογραφή)

.....

Κιμωνίδης Αλέξανδρος
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.
©2021 - All rights reserved.

Περίληψη

Ο όγκος των φορτίων εργασίας που εκτελούνται στο Cloud αυξάνεται συνεχώς. Οι φορείς εκμετάλλευσης κέντρων δεδομένων και οι πάροχοι cloud έχουν υιοθετήσει τη συνεγκατάσταση φορτίων εργασίας και την πολλαπλή μίσθωση ως πρώτης τάξεως ζητήματα σχεδιασμού συστημάτων για την αποτελεσματική εξυπηρέτηση και διαχείριση αυτών των τεράστιων υπολογιστικών αναγκών. Οι σημερινοί υπερσύγχρονοι διαχειριστές πόρων τοποθετούν τις εφαρμογές στη διαθέσιμη δεξαμενή πόρων χρησιμοποιώντας τυποποιημένες μετρήσεις, όπως η χρήση της CPU ή της μνήμης. Ως αποτέλεσμα, οι σημερινοί διαχειριστές πόρων αποτυγχάνουν να επιτύχουν επαρκή χρήση των πόρων.

Στην παρούσα διατριβή, σχεδιάζουμε έναν διαχειριστή πόρων που αξιοποιεί τη βαθιά ενισχυτική μάθηση για την πολιτική του και χρησιμοποιεί μετρητές παρακολούθησης της απόδοσης, οι οποίοι είναι μια πιο σύνθετη μετρική που είναι σε θέση να προσδιορίσει την τρέχουσα κατάσταση μιας μηχανής. Παρουσιάζουμε τον αντίκτυπο της εφαρμογής πίεσης σε διαφορετικούς πόρους διακομιστών και την ανάγκη για έναν καλύτερο χρονοπρογραμματιστή που λαμβάνει υπόψη του τις σωστές μετρικές. Ενσωματώνουμε τη λύση μας με το OpenAI Gym, ένα από τα πιο ευρέως χρησιμοποιούμενα πακέτα εργαλείων για την ανάπτυξη και τη σύγκριση αλγορίθμων ενισχυτικής μάθησης, και δείχνουμε ότι μπορούμε να επιτύχουμε υψηλότερη χρήση πόρων σε σύγκριση με τον προεπιλεγμένο χρονοπρογραμματιστή καθώς και με άλλους χρονοπρογραμματιστές τελευταίας τεχνολογίας.

Λέξεις Κλειδιά— υπολογιστικό νέφος, διαχείριση πόρων, βαθιά ενιχυμένη μάθηση, μετρητές παρακολούθησης επιδόσεων

Abstract

The amount of workloads ran on the Cloud is growing all the time. Data center operators and cloud providers have embraced workload co-location and multi-tenancy as first-class system design concerns to efficiently service and manage these massive computing needs. Current state-of-the-art resource managers place applications on the available pool of resources using standard metrics such as CPU or memory usage. As a result, current state-of-the-art resource managers fail to achieve adequate resource utilization.

In this thesis, we design a resource manager that leverages deep reinforcement learning for its policy and uses performance monitoring counters which are a more complex metric that is able to determine a machine's current state. We showcase the impact of applying stress on different server resources and the need for a better scheduler that considers the correct metrics. We integrate our solution with OpenAI Gym, one of the most widely used tool-kits for developing and comparing reinforcement learning algorithms, and we show that we can achieve higher resource usage compared to the default scheduler as well as other state-of-the-art schedulers.

Keywords— cloud computing, resource management, deep reinforcement learning, scheduling, performance monitoring counters

Ευχαριστίες

Αρχικά, θα ήθελα να εκφράσω την ευγνωμοσύνη μου στον επιβλέποντα μου, Καθηγητή Δημήτριο Σούντρη ΕΜΠ, ο οποίος με εμπιστεύτηκε και μου έδωσε την ευκαιρία να εκπονήσω τη διπλωματική μου εργασία στο Εργαστήριο Μικροϋπολογιστών και Ψηφιακών Συστημάτων (MicroLab) στο ΕΜΠ.

Επίσης, θα ήθελα να ευχαριστήσω τον μεταδιδακτορικό ερευνητή Σωτήριο Ξύδη, τον υποψήφιο διδάκτορα Δημοσθένη Μασούρο και τον Dr. Rajiv Nishtala για τη βοήθεια και τη συνεργασία τους καθ΄όλη τη διάρκεια της διπλωματικής μου. Η συνεχής τριβή μας κατά τη διάρκεια της διπλωματικής, με βοήθησε να αποκτήσω γρηγορότερα γνώσεις σχετικές με το αντικείμενο που εξετάσαμε, οι οποίες ταυτόχρονα είναι εφαρμόσιμες σε πολλούς τομείς της σύγχρονης τεχνολογίας και των συστημάτων υπολογιστών. Παράλληλα με εισήγαγαν στην ερευνητική προσέγγιση και εργασία. Θα ήθελα επίσης να ευχαριστήσω όλα τα μέλη του Microlab για το ευχάριστο περιβάλλον εργασίας.

Acknowledgments

Firstly, I would like to express my gratitude to my supervisor, Professor Dimitrios Soudris NTUA, who trusted me and gave me the opportunity to develop my thesis at the Microprocessors and Digital Systems Laboratory (MicroLab) in National Technical University of Athens (NTUA). In addition, I would like to thank the Post-Doctoral Researcher and Proferssor Sotirios Xydis, the Ph.D candidate Dimosthenis Masouros and Dr. Rajiv Nishtala for their assistance and cooperation throughout my diploma thesis development. Our continuous collaboration during this thesis helped me to gain useful knowledge regarding to the subject we examined, which is at the same time applicable to many areas of modern technology and computer engineering. They also introduced me to the research approach and environment. Furthermore, I would like to thank all the members of MicroLab for a pleasant working environment.

Contents

List of Figures

List of Tables

Εκτεταμένη Περίληψη

1 Εισαγωγή

Την τελευταία δεκαετία, η υιοθέτηση των υπολογιστικών συστημάτων νέφους παρουσίασε σημαντική ανάπτυξη, τόσο σε επίπεδο καταναλωτών όσο και επιχειρήσεων, και θα συνεχίσει να εξελίσσεται στο μέλλον. Η εξέλιξη και η προσφορά της τεχνολογίας εικονικοποίησης virtualization βασισμένης σε πακέτα (containers) , καθώς και τα πλεονεκτήματα που προσφέρουν οι υπολογιστές νέφους στους χρήστες και στους διαχειριστές, έχουν αποτελέσει έναυσμα προς αυτή την κατεύθυνση. Οι χρήστες έχουν τη δυνατότητα να εκτελέσουν διαφορετικά είδη εφαρμογών και υπηρεσιών, πληρώνοντας μόνο τους πόρους που χρησιμοποιούνται σε μια δεδομένη στιγμή, ενώ παράλληλα επιτρέπεται η ανάπτυξη οικονομιών κλίμακας για τους φορείς εκμετάλλευσης των πόρων νέφους, οι οποίοι τους διαμοιράζουν σε διαφορετικούς χρήστες. Η αύξηση του όγκου του φόρτου εργασιών που φορτώθηκαν και εκτελέστηκαν σε υπολογιστές νέφους, έχουν αναγκάσει τους φορείς εκμετάλλευσης των κέντρων δεδομένων και τους παρόχους υπηρεσιών νέφους, όπως το Google Cloud Platform και Amazon EC2 (AWS) να θέσουν ως σημαντική προτεραιότητα τους το σχεδιασμό ενός συστήματος με γνώμονα την συντοποθέτηση εφαρμογών, καθώς επίσης και τον διαμοιρασμό πόρων μεταξύ διαφορετικών χρηστών.

Ωστόσο, αυτή η κατανομή πόρων μεταξύ ξεχωριστών χρηστών δεν έρχεται έτοιμη. Τα φορτία εργασίας που τοποθετούνται σε κοινά φυσικά μηχανήματα, ανταγωνίζονται συνεχώς για κοινόχρηστους πόρους, όπως η κρυφή μνήμη, η χρήση της κεντρικής μνήμης, το εύρος ζώνης του δικτύου και της μνήμης και άλλους, προκαλώντας τεράστιες αρνητικές επιδράσεις στην απόδοση. Η κατάσταση αυτή εξελίσσεται καθώς οι νέοι προμηθευτές υπηρεσιών υπολογιστών νέφους, προσφέρουν στους χρήστες ελαστικότητα και τη δυνατότητα γρήγορης και εύκολης ανανέωσης της χωρητικότητας των μηχανών που ενοικιάζουν, οδηγώντας σε έναν δυναμικό εφοδιασμό των πόρων συστήματος.

Αυτή η ευελιξία στη δυνατότητα κλιμάκωσης των πόρων οδήγησε τους χρήστες να ζητούν όλο και περισσότερους πόρους, ώστε να ικανοποιήσουν τις απαιτήσεις τους σχετικά με την ποιότητα των υπηρεσιών που παρέχουν οι ευαίσθητες στην καθυστέρηση εφαρμογές τους. Ωστόσο, ακόμη και σε μεγάλες εταιρίες όπως η Microsoft και η Google η μέση χρήση των διαθέσιμων πόρων είναι συνήθως κάτω

από 50%. Επιπλέον, τα κέντρα δεδομένων της Mozilla και της VMWare λειτουργούν με 6% και 20-30% χρήση αντίστοιχα. Οι πάροχοι υπηρεσιών διαδικτύου έχουν προσδιορίσει ως κρίσιμο στόχο σχεδιασμού τη βελτίωση της χρήσης των σύγχρονων υπολογιστών αποθήκευσης με σκοπό τη μείωση του συνολικού κόστους ιδιοκτησίας. Από την άλλη πλευρά, τα πράγματα γίνονται ακόμη χειρότερα, στους διαχειριστές και ενορχηστρωτές σε συστοιχίες υπολογιστών που επιτρέπουν διαμοιρασμό του συστήματος μεταξύ διαφορετικών ομάδων εφαρμογών. Χαρακτηριστικά σε μια συστοιχία υπολογιστών του Twitter η χρήση των διαθέσιμων πυρήνων ήταν κάτω από 20%, ενώ την ίδια στιγμή, οι δεσμευμένοι πόροι φτάνουν μέχρι και το 80% της συνολικής χωρητικότητας. Οι διαχειριστές μεγάλων κέντρων υπολογιστών αποτυγχάνουν να εξασφαλίσουν την κατάλληλη ποσότητα πόρων. Τέλος ο ΄ώριμος΄ διαχειριστής συστοιχιών Borg επιτυγχάνει 25-35% και 40% χρήση επεξεργαστών και μνήμης αντίστοιχα, ενώ οι δεσμευμένοι πόροι είναι την ίδια στιγμή 75% και 60% αντίστοιχα.

Επιπλέον, η συνεχής εξέλιξη των τεχνολογιών και των γενεών υλικού, απαιτεί από τους φορείς εκμετάλλευσης των κέντρων δεδομένων να αναβαθμίζουν επανειλημμένα την υποκείμενη υποδομή τους, προκειμένου να συμβαδίσουν με τις τελευταίες εξελίξεις και να επιτρέπουν στους παρόχους υπηρεσιών νέφους να παρέχουν καλύτερη ποιότητα υπηρεσιών, οδηγώντας σε συστοιχίες με διαφορετικές, ανομοιογενείς διαμορφώσεις διακομιστών. Από τα παραπάνω, είναι προφανές ότι η συμφόρηση και η πολυδιάστατη φύση διαφόρων διαμοιραζόμενων πόρων μπορούν να προκαλέσουν σημαντική επίπτωση στην αποδοση των εκτελούμενων εφαρμογών και επομένως αναδύεται η ανάγκη μιας ενήμερης σχετικά με τον ανταγωνισμό που υπάρχει για τη χρήση των πόρων αλλα και την πιθανή ετερογένεια εντός μιας συστοιχίας εφαρμογών, δρομολόγησης εισερχόμενων εκτελέσιμων φορτίων.

Η τρέχουσα τάση, στους οργανισμούς, για την δρομολόγηση των εισερχόμενων εφαρμογών σε ένα σύνολο διαθέσιμων πόρων είναι μέσω ενορχηστρωτων (container orchestrators) , όπως είναι το Kubernetes ή το Mesos. Η εξέλιξη και οι βελτιώσεις στην απόδοση που επέφερε η εικονικοποίηση (virtualization) των εφαρμογών, οδήγησε τις εταιρίες να αλλάξουν τον τρόπο με τον οποίο αναπτύσσουν τις εφαρμογές τους σε προσαρμοσμένες σε περιβάλλον υπολογιστών νέφους μικρο-υπηρεσίες με χρήση containers. Ωστόσο οι υλοποιήσεις τέτοιων ενορχηστρωτών containers έχουν σχεδιαστεί με γνώμονα την απομόνωση πόρων και όχι απαραίτητα την αποδοτικότερη χρήση αυτών. Φαντάζει επιτακτική λοιπόν η ανάγκη για έναν δρομολογητή σε περιβάλλοντα νέφους, οποίος θα στοχεύει παράλληλα στην μεγιστοποίηση του ποσοστού χρήσης των υπάρχοντων πόρων αλλά και της απόδοσης των εφαρμογών που εκτελούνται.

2 Ιστορικό για την βαθιά ενισχυτική μάθηση και τεχνολογίες κατανομής

Η μηχανική μάθηση είναι ένας ευρύς όρος που αναφέρεται σε αυτοματοποιημένες μεθόδους υπολογιστών που μαθαίνουν μια εργασία από ένα σύνολο παραδειγμάτων και βασίζονται σε λογικές ή δυαδικές πράξεις.

2.1 Βαθιά Ενισχυτική Μάθηση

Η βαθιά ενισχυτική μάθηση είναι ένας τύπος μηχανικής μάθησης που έχει αποκτήσει μεγάλη δημοτικότητα τα τελευταία χρόνια, λόγω των απίστευτων αποτελεσμάτων του σε μια ποικιλία εφαρμογών όπως αναγνώριση μοτίβου, αναγνώριση ήχου, όραση υπολογιστή και επεξεργασία φυσικής γλώσσας . Οι προσεγγίσεις βαθιάς μάθησης μπορούν επίσης να συνδυαστούν με ενισχυτικές μεθόδους εκμάθησης για την ανάπτυξη ουσιαστικών αναπαραστάσεων για θέματα με υψηλό επίπεδο διαστατικά εισαγόμενα ακατέργαστα δεδομένα, σύμφωνα με πρόσφατη μελέτη.

Αλγόριθμοι Ενισχυτικής Μάθησης

Η Ενισχυτική Μάθηση(RL) είναι ένας τύπος μηχανικής μάθησης στην οποία ένας υπολογιστής μαθαίνει αλληλεπιδρώντας με το περιβάλλον του. Ένας πράκτορας μπορεί να μάθει μέσω δοκιμής και σφάλματος χρησιμοποιώντας ένα RL framework. Ο στόχος του πράκτορα είναι να μάθει να επιλέγει συμπεριφορές που μεγιστοποιούν την προβλεπόμενη αθροιστική ανταμοιβή με την πάροδο του χρόνου αλληλεπιδρώντας με το περιβάλλον. Με άλλα λόγια, ο πράκτορας επιδιώκει να μάθει μια ιδανική ακολουθία δραστηριοτήτων που πρέπει να εκτελεστούν προκειμένου να επιτευχθεί ο στόχος του παρακολουθώντας τα αποτελέσματα των δράσεων που κάνει στο περιβάλλον.

Μια διαδικασία απόφασης Markov(MDP) μπορεί να χρησιμοποιηθεί για να προσομοιώσει έναν RL πράκτορα. Το ζήτημα ονομάζεται πεπερασμένο MDP εάν οι καταστάσεις και οι χώροι δράσης είναι πεπερασμένοι. Τα πεπερασμένα MDP είναι κρίσιμα για θέματα RL και μεγάλο μέρος της βιβλιογραφίας υποθέτει ότι το περιβάλλον είναι ένα πεπερασμένο MDP.

Βαθιά Μάθηση

Η ικανότητα μιας προσέγγισης μηχανικής μάθησης να αποδίδει καλά εξαρτάται έντονα απο την ποιότητα της αναπαράστασης των δεδομένων εισόδου. Ως αποτέλεσμα, η προεπεξεργασία δεδομένων είναι ένα σημαντικό στάδιο.

Χωρίς ανθρώπινη εμπειρογνωμοσύνη, η αναπαράσταση βασίζεται σε δεδομένα που τίθενται απευθείας σε βαθιά δίκτυα (δηλαδή, αυτόματη εξαγωγή χαρακτηρισ-

τικών). Αυτό το σημαντικό χαρακτηριστικό των αρχιτεκτονικών βαθιάς μάθησης έχει βοηθήσει την πρόοδο προς τους αλγόριθμους που είναι ο στόχος της Τεχνητής Νοημοσύνης (AI), επιτρέποντάς της να κατανοήσει τον κόσμο γύρω του χωρίς να χρειάζεται εξειδικευμένη γνώση ή ανάμειξη.

Πολλαπλά επίπεδα αναπαραστάσεων χρησιμοποιούνται σε μοντέλα βαθιάς εκμάθησης. Στην πραγματικότητα ειναι μια συλλογή εξαρτημάτων συμπεριλαμβανομένων των αυτόματων κωδικοποιητών, Restricted Boltzmann Μηχανημάτων (RBM), και συνελικτικές στρώσεις. Τα ακατέργαστα δεδομένα εισάγονται σε ένα δίκτυο με πολλά επίπεδα κατά τη διάρκεια της προπόνησης. Οι έξοδοι κάθε επιπέδου, οι οποίες είναι μη γραμμικοί μετασχηματισμοί χαρακτηριστικών, χρησημοποιούνται ως είσοδοι στα επόμενα επίπεδα του βαθύ δικτύου. Στο τελευταίο στρώμα η αναπαράσταση εξόδου μπορεί να χρησιμοποιηθεί για τον περιορισμό ταξινομητών ή εφαρμογών που επωφελούνται από αφηρημένες αναπαραστάσεις δεδομένων με ιεραρχικό τρόπο ως εισροές. Κάθε επίπεδο προσπαθεί να μάθει και να αποκαλύψει τα υποκείμενα επεξηγηματικά στοιχεία εφαρμόζοντας έναν μη γραμμικό μετασχηματισμό στην είσοδό του. Ως αποτέλεσμα, αυτή η διαδικασία οδηγεί στο σχηματισμό μιας ιεραρχίας αφηρημένων αναπαραστάσεων.

Οι προσεγγίσεις βαθιάς μάθησης μπορούν τώρα να χρησιμοποιηθούν για την επίλυση προβλημάτων του πραγματικού κόσμου χάρη στη χρήση βαθιών νευρωνικών δικτύων. Ωστόσο, μαθαίνοντας τις παραμέτρους σε μια βαθιά αρχιτεκτονική είναι μια δύσκολη διαδικασία βελτιστοποίησης με υψηλή υπολογιστική πολυπλοκότητα (τα βαθιά δίκτυα με πολλαπλά κρυμμένα επίπεδα έχουν εκατομμύρια παραμέτρους για μάθηση). Ευτυχώς, η εμφάνιση εξελιγμένων τεχνολογιών παράλληλης επεξεργασίας όπως τα GPU έχει αντιμετωπίσει αυτή τη δυσκολία σε κάποιο βαθμό.

2.2 Τεχνολογία κατανομής μνήμης Cache

Η τεχνολογία Cache Allocation είναι ένας τρόπος διαχωρισμού της κρυφής μνήμης L3 σε τμήματα και διαχωρισμού τμημάτων μεταξύ τους. Συγκεκριμένα, η τεχνολογία κατανομής κρυφής μνήμης μας επιτρέπει να ορίσουμε κλάσεις υπηρεσιών(CLOS) στις οποίες μπορούμε μετά να αντιστοιχίσουμε πυρήνες και με αυτόν τον τρόπο επιτυγχάνουμε την κατάτμηση της κρυφής μνήμης. Οι CLOS μπορούν επίσης να μοιράζονται ways κρυφής μνήμης μεταξύ τους. Οι προαναφερθείσες διεπαφές για τις CLOS υλοποιούνται με τη χρήση Model-Specific-Registries(MSRs). Κάθε CLOS έχει μια μάσκα η οποία ορίζει τα cache ways που περιέχει, και αυτα τα cache ways πρέπει να είναι διαδοχικά.

2.3 Περιορισμός πυρήνων διεργασίας

Το Psutil (python system and process utilities) είναι μια βιβλιοθήκη πολλαπλών πλατφορμών για την ανάκτηση πληροφοριών σχετικά με τις εκτελούμενες διερ-

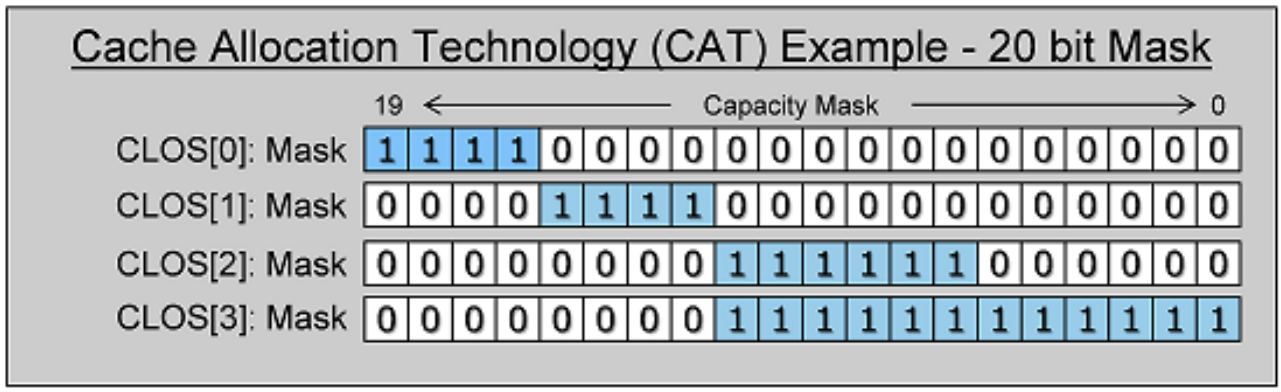

Figure 1: Example of Capacity Bitmask (CBM) overlap and isolation across multiple Classes of Service (CLOS)

γασίες και τη χρήση του συστήματος (CPU, μνήμη, δίσκοι, δίκτυο, αισθητήρες) σε Python. Είναι χρήσιμη κυρίως για την παρακολούθηση του συστήματος, τη δημιουργία προφίλ, τον περιορισμό των πόρων των διεργασιών και τη διαχείριση των διεργασιών που εκτελούνται. Υλοποιεί πολλές λειτουργίες που προσφέρονται από εργαλεία γραμμής εντολών UNIX όπως: ps, top, lsof, netstat, ifconfig, who, df, kill, free, nice, ionice, iostat, iotop, uptime, pidof, tty, taskset, pmap. psutil. Στην υλοποίηση του Linux το psutil χρησιμοποιεί το taskset για τον περιορισμό των πυρήνων διεργασιών.

3 Πειραματική ρύθμιση και χαρακτηρισμός φόρτου εργασίας

Σε αυτό το κεφάλαιο αξιολογούμε τη συμπεριφορά των εφαρμογών ενός πυρήνα σε σχέση με αλλαγές στη μνήμη cache και τη συμπεριφορά των εφαρμογών cloud πολλαπλών νημάτων όσον αφορά το αυξημένο φορτίο έτσι ώστε να βρεθεί το γόνατο της εφαρμογής.

3.1 Χαρακτηριστικά Συστήματος

Για το υπόλοιπο αυτής της εργασίας θα χρησιμοποιούμε το σύστημα στο πίνακα 1.

3.2 Χαρακτηρισμός εφαρμογών ενός πυρήνα σε σχέση με αλλαγές στη μνήμη cache

Τα σχήματα στο Figures 2 βρέθηκαν τρέχωντας benchmarks απο το SPEC. Για το υπόλοιπο της εργασίας θα ονομάζουμε cache sensitive της εφαρμογές που παρουσιάζουνε μεγάλη αλλαγή στην απόδοση οταν αλλάζουμε την μνήμη cache.

Table 1: Platform Specification

Model	Intel(R) Xeon(R) CPU E5-2658A v3 @ 2.20GHz
OS	Ubuntu 18.04 (kernel 4.15)
Sockets	2
Cores/Socket	12
Threads/Core	2
Base/Max Turbo Frequency	2.2GHz / 2.9GHz
L1 Inst/Data Cache	32 / 32 KB
L2 Cache	256 KB
L3 (Last-Level) Cache	31 MB
Memory	16GBx8, 2400MHz DDR4

Table 2: Single Core Application Characterization

Application Name	IPC (2 ways)	IPC (20 ways)	Percentage Difference
gcc	0.95	1.23	29.47
soplex	1.02	1.12	9.8
sphinx	1.55	2.05	32.26
xalancbmk	1.15	1.65	43.48

3.3 Χαρακτηρισμός εφαρμογών cloud πολλαπλών νημάτων σε σχέση με αυξημένο φορτίο

Σε αυτήν την ενότητα αξιολογούμε τη συμπεριφορά των εφαρμογών cloud πολλαπλών νημάτων από τη σουίτα συγκριτικής αξιολόγησης Tailbench[1]. Τα benchmark που χρησημοποιήθηκαν ήταν sphinx, img-dnn, xapian, masstree.

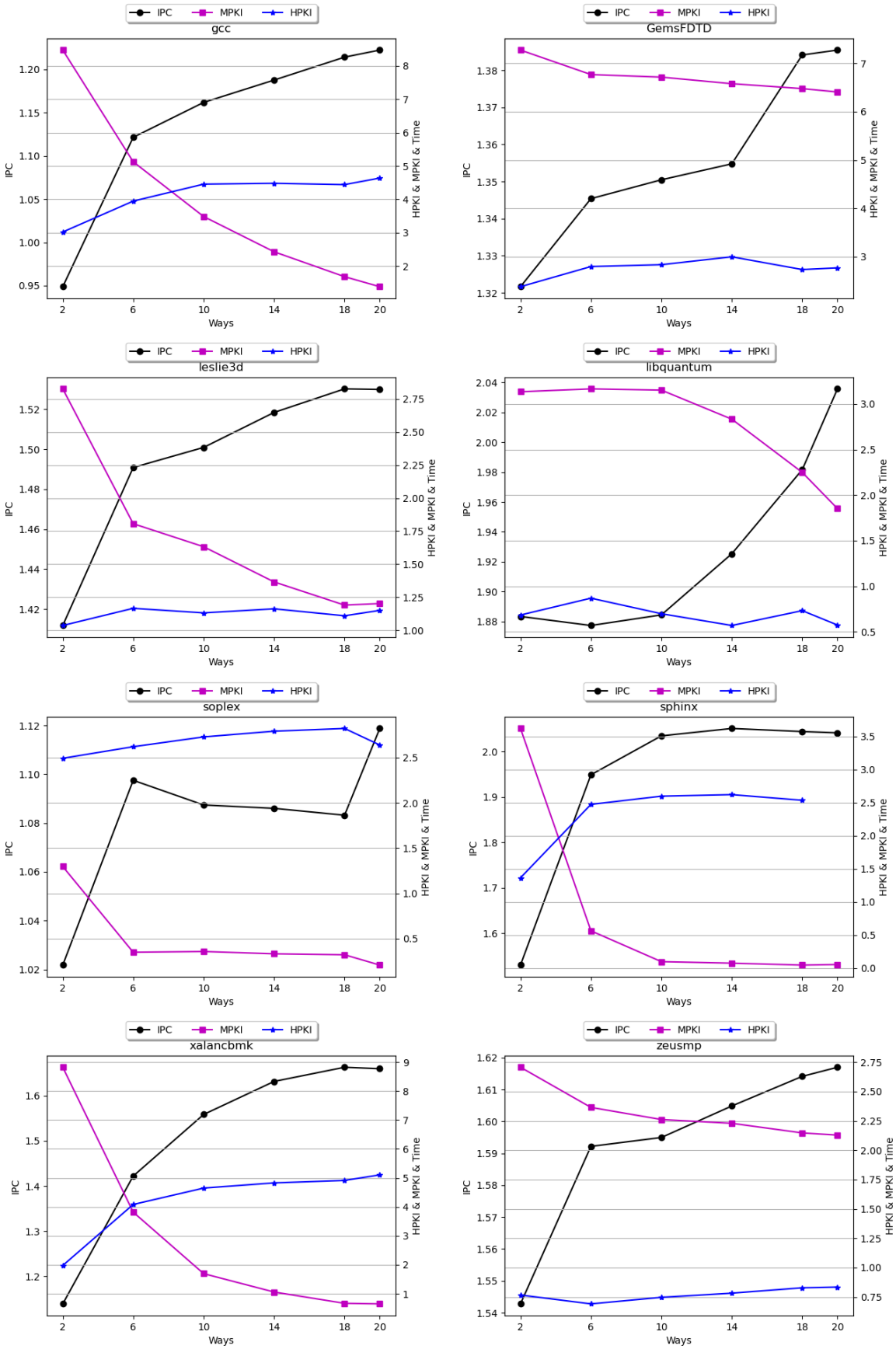

Figure 2: (a) gcc (b) GemsFDTD (c) Leslie3d (d) libquantum (e) soplex (f) sphinx (g) xalancbmk (h) zeusmp

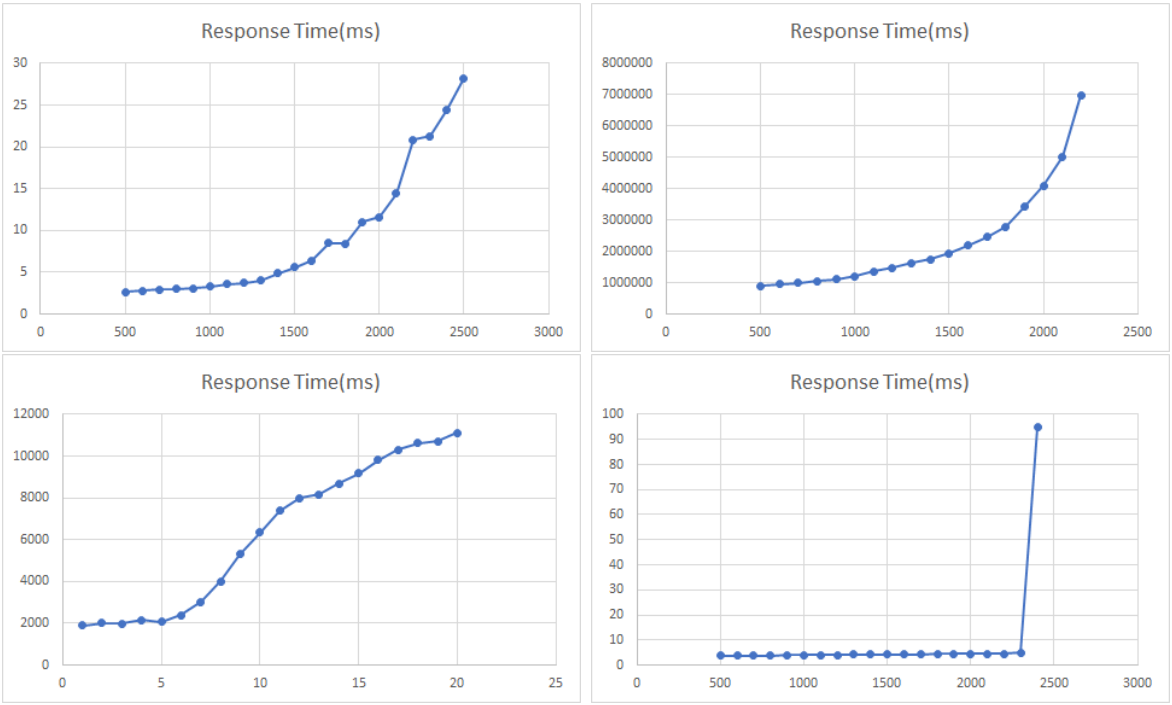

Figure 3: (a) img-dnn (b) masstree (c) sphinx (d) xapian
Tail latency with increasing input load (RPS). The vertical lines show the knee of each curve, which is hereafter referred to as maxload. The horizontal lines show the latency at max load, which is used to determine the QoS targets of each application (detailed numbers can be found in Table 4.3).

Έχοντας τα παραπάνω σχήματα βρήκαμε για κάθε εφαρμογή το 'γονατό' της, δηλαδή το σημείο στο οποίο ο ρυθμός αύξησης τον χρόνου εξυπηρέτησης γίνεται εκθετικός. Το γόνατο της κάθε εφαρμογής μας δίνει το μέγιστο φόρτο εργασίας που μπορεί να δεχτεί η κάθε εφαρμογή και επίσης μας δίνει το χρόνο εξυπηρέτησης που θα θέλαμε για το αναφερόμενο φόρτο εργασίας.

Application Name	Max Load under QoS (RPS)	Target QoS (msec)
sphinx	5	2000
masstree	1000	1.2
img-dnn	1300	4
xapian	2300	4.8

Table 3: Latency-Critical Applications

4 Διαχειριστής Πόρων

Αυτή η ενότητα εισάγει τον δικό μας Διαχειριστή Πόρων, μια λύση βασισμένη στη βαθιά ενισχυτική μάθηση για την κατανομή πόρων των κρίσημων υπηρεσιών. Ο ελεγκτής αξιοποιεί μια απλή DQN αρχιτεκτονική που λαμβάνει PMC ως είσοδο και λαμβάνει λεπτομερείς αποφάσεις για την κατανομή των πόρων. Ο σχεδιαστικός στόχος του ελεγκτή είναι να ελαχιστοποιήσει την χρήση των πόρων επιτυχόντας

τους στόχους χρόνου εξυπηρέτησης της υπηρεσίας.

4.1 Αρχές Σχεδιασμού

- Οι αποφάσεις για την κατανομή των πόρων είναι δυναμικές και λεπτές. Οι LC εφαρμογές είναι πολύ ευαίσθητες στις εκχωρήσεις πόρων. Με λεπτομερείς αλλαγές πόρων ελαχιστοποιούμε το χρόνο που χάνεται στον προγραμματισμό των thread και το αντίκτυπο των αλλαγών σταθερής εκχώρησης πόρων στη μνήμη cache.

- Δεν απαιτείται εκ των προτέρων γνώση εφαρμογής ή/και δημιουργία προφίλ offline.Η δημιουργία προφίλ offline σε όλες τις πιθανές συνθέσεις εφαρμογών, ακόμη και αν είναι εφικτό, θα ήταν απαγορευτικά δαπανηρή. Επιπλέον, η απόκτηση αυτών των πληροφοριών δεν είναι πάντα δυνατή, ιδίως στο πλαίσιο της δημόσιας υπηρεσίας νέφους που φιλοξενεί άγνωστους φόρτους εργασίας.

- Ο ελεγκτής ανακτά γρήγορα από εσφαλμένες αποφάσεις.Καθώς ο διαχειριστής πόρων εξερευνά το χώρο εκχώρησης online, αναπόφευκτα ορισμένες από τις αποφάσεις της ενδέχεται να είναι αντιπαραγωγικές. Αλλά το νευρικό δίκτυο μαθαίνει αρκετά γρήγορα ώστε να ελαχιστοποιεί τέτοιες καταστάσεις.

4.2 Resource Manager Architecture

4.3 Deep-Q-Network(DQN) Agent

Ο πράκτορας της DQN μαθαίνει τις ̇βέλτιστες' αποφάσεις με την πάροδο του χρόνου, αλληλεπιδρώντας με το περιβάλλον με τη χρήση του διλήμματος εξερεύνησης-εκμετάλλευσης. Στο δίλημμα εξερεύνησης-εκμετάλλευσης, ο πράκτορας όχι μόνο συλλαμβάνει την ανάγκη εκμετάλλευσης της ̇βελτιστής' λύσης που έχει βρεθεί μέχρι στιγμής αλλά διερευνά επίσης ενέργειες που μπορεί να είναι ή όχι καλύτερες. Η πιθανότητα εξερεύνησης αντί εκμετάλλευσης λαμβάνεται από το παράγοντα έψιλον. Ενώ η ύπαρξη ενός σταθερού αλλά μικρού ε είναι η κυρίαρχη προσέγγιση σε καθαρά RL ρυθμίσεις, γίνεται ανέφικτο σε μεγάλους χώρους δράσης. Ο διαχειριστής πόρων χρησιμοποιεί αντίθετα την έψιλον προσάρτηση, η οποία μεταβαίνει από μια διερευνητική πολιτική σε μια εκμεταλλευτική πολιτική με την πάροδο του χρόνου για την αποτελεσματική εξερεύνηση του τομέα διακριτής δράσης. Οι αλληλεπιδράσεις του πράκτορα με το περιβάλλον σε κάθε στάδιο καθοδηγούνται από την συγκέντρωση της κατάστασης, δημιουργώντας ενέργειες που είναι είτε καθοριστικές είτε τυχαίες. Στη συνέχεια, ο παράγοντας λαμβάνει μια αμοιβή στο επόμενο βήμα χρόνου που καθορίζει πόσο καλα τα πήγε ο πράκτορας στο προηγούμενο βήμα. Ο διαχειριστής πόρων επιλύει τη διαχείριση εργασιών με τη μετάφρασή του σε ένα πρόβλημα απόφασης Markov (MDP) που κατόπιν επιλύθηκε από το DQN. Ο DQN υλοποιείται χρησιμοποιώντας Stable Baselines 2 και όλο

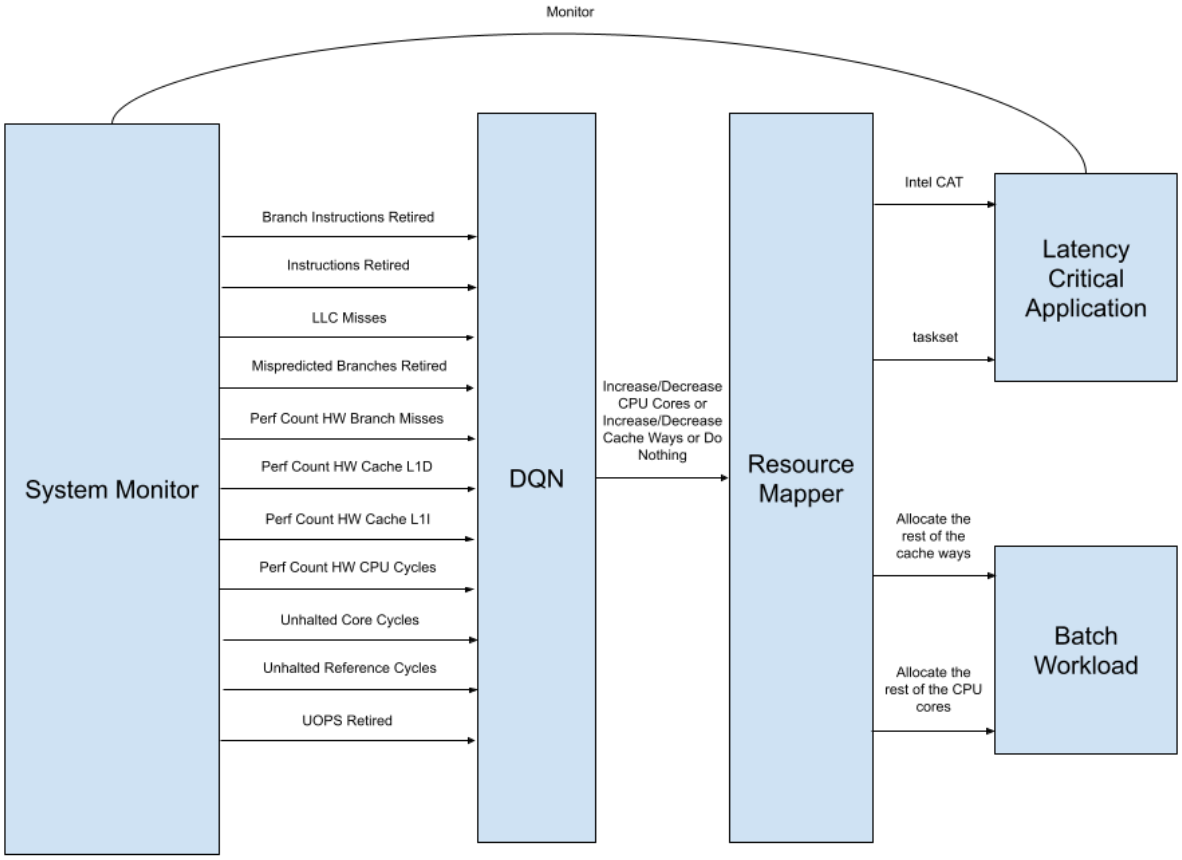

Figure 4: An overview of the controller's architecture

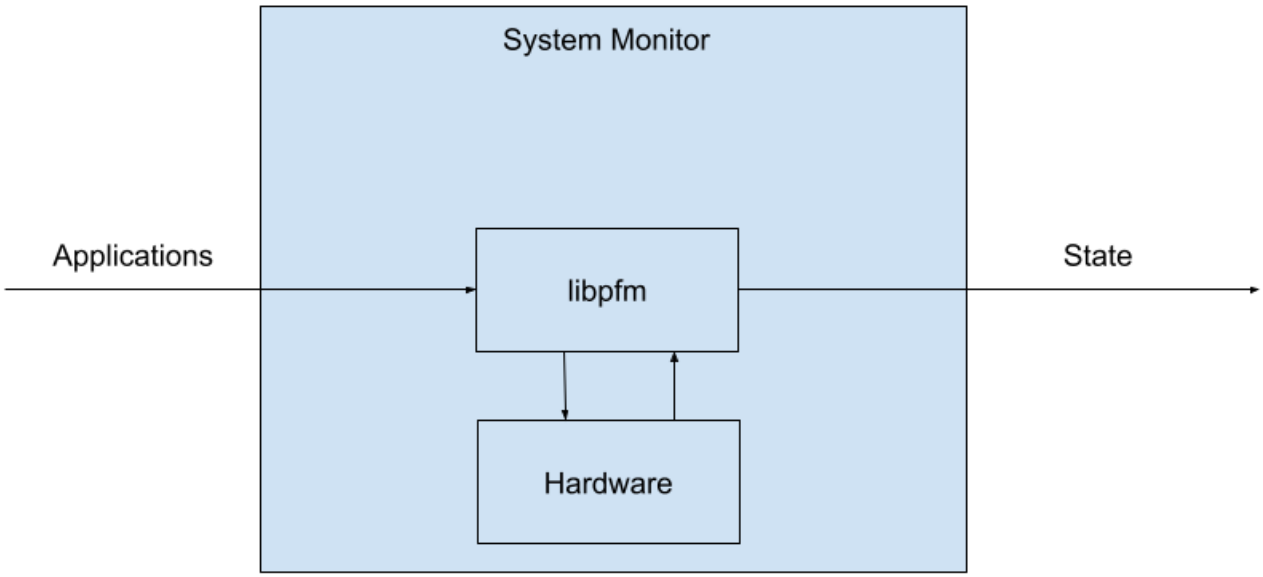

Figure 5: A basic overview of the system monitor

το περιβάλλον του διαχειριστή πόρων ενσωματώθηκε σε ένα περιβάλλον OpenAI Gym ώστε να γίνει όσο το δυνατόν πιο απλο και αρθρωτό.

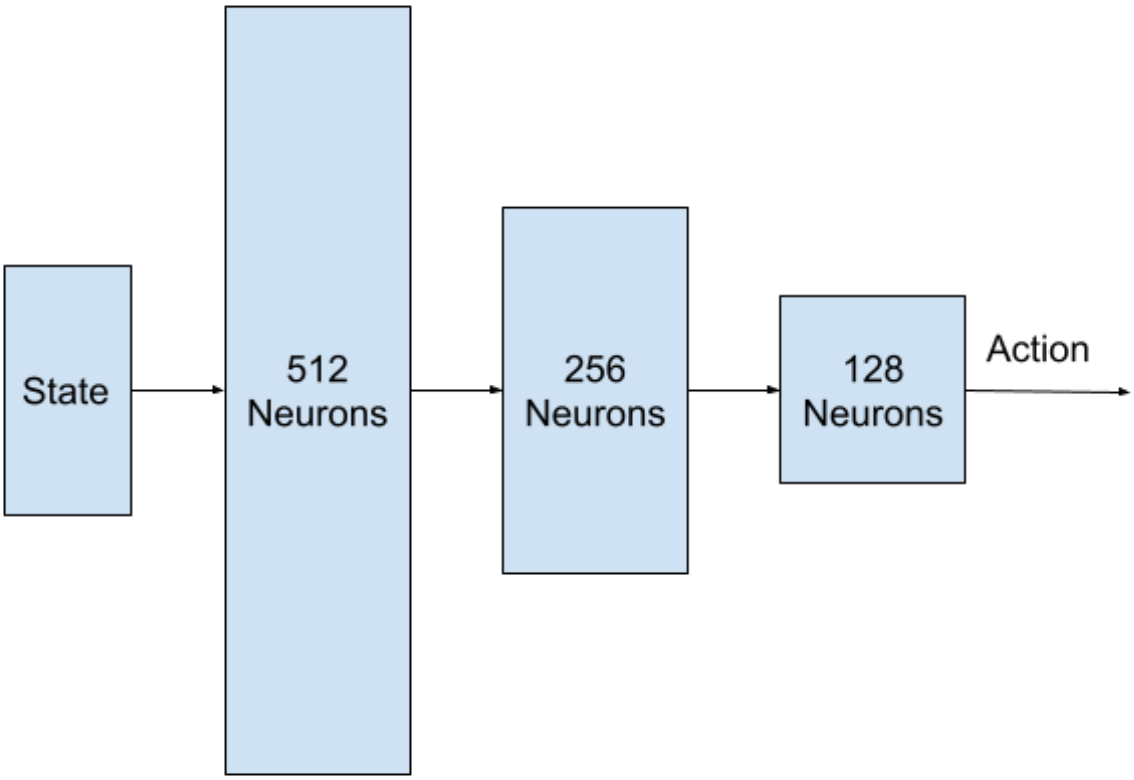

Figure 6: A basic overview of the neural network's architecture

Reward Function

Ο μηχανισμός επιβράβευσης του διαχειριστή πόρων καθορίζει τις αποφάσεις αντιστοίχισης που βασίζονται σε PMC και καλούνται περιοδικά σε κάθε χρονικό διάστημα παρακολούθησης. Η ανταμοιβή αποσκοπεί στην ελαχιστοποίηση της χρήσης πόρων που υπόκειται στην εκπλήρωση του στόχου QoS, και εκφράζεται ως εξής:

$$reward = \begin{cases} \frac{QoS_{target}}{QoS} + \frac{20}{currentCacheWays} + \frac{24}{currentCores} & , QoS \leq QoS_{target} \\ max(-(QoS/QoS_{target})^3, -\phi) & , QoS > QoS_{target} \end{cases} \quad (1)$$

4.4 Resource Mapper

Ο Resource Mapper έχει τρεις βασικούς ρόλους:

- Να λαμβάνει αιτήσεις εκχώρησης πόρων.
- Να βεβαιώνεται ότι τα νήματα της εφαρμογής έχουν αντιστοιχιστεί στους πυρήνες και τα cache ways.
- Να εκχωρεί τυχόν μη χρησιμοποιημένους πόρους στα batch workloads

Αυτό πραγματοποιήθηκε με τη χρήση της εντολής taskset των Linux για την αντιστοίχηση νημάτων σε πυρήνες και της τεχνολογίας CAT της Intel για την αντιστοίχιση cache ways στα CLOS που εκτελείται η εφαρμογή. Εάν εκτελούμε πολλές εφαρμογές ταυτόχρονα, τότε η πολιτική που χρησιμοποιεί ο ελεγκτής ειναι first come first served.

4.5 Υλοποίηση του Διαχειριστή Πόρων

Table 4: Performance Monitoring Counters

#	PMC
1	UNHALTED CORE CYCLES
2	INSTRUCTION RETIRED
3	PERF COUNT HW CPU CYCLES
4	UNHALTED REFERENCE CYCLES
5	UOPS RETIRED
6	BRANCH INSTRUCTIONS RETIRED
7	MISPREDICTED BRANCH RETIRED
8	PERF COUNT HW BRANCH˙MISSES
9	LLC MISSES
10	PERF COUNT HW CACHE L1D
11	PERF COUNT HW CACHE L1I

Ο διαχειριστής πόρων υλοποιείται στο χώρο χρήστη και χρησιμοποιεί μόνο υποστήριξη υλικού που έχει εκτεθεί από τον πυρήνα Linux. Τα στοιχεία που εξαρτώνται από το υλικό είναι τα PMC.

Μέτρηση PMCs. Οι PMC μετρώνται με τη χρήση του εργαλείου παρακολούθησης perfmon (libpfm 4.10.0). Ο πίνακας 5.1 παρουσιάζει τα PMC που επιλέξαμε για είσοδο στο DQN. Η μέγιστη τιμή για τους μετρητές 1-5 λήφθηκε με την εκτέλεση μικροδείκτη έντασης CPU που αποτελείται από αρκετές μαθηματικές λειτουργίες χωρίς προσβάσεις μνήμης· για τους μετρητές 6-8 εκτελώντας ένα μιςροβενςημαρκ που προκαλεί πολυάριθμα σφάλματα κλάδου συγκεντρώνοντας στοιχεία από ένα μη ταξινομημένο διάνυσμα δεδομένων για να ελέγξει εάν είναι μεγαλύτερα από ορισμένη αξία· και για τους μετρητές 9-11 ελήφθησαν με την εκτέλεση του δείκτη αναφοράς ροής.

Μέτρηση QoS. Ως απόδειξη της ιδέας, ο χρόνος απόκρισης LC μετριέται με την καταγραφή της εξόδου από την υπηρεσία LC η οποία εξάγει την απόκρισή του

προγράμματος σε σταθερό διάστημα σταθμοσκόπησης. Μια εναλλακτική λύση θα ήταν η συγκέντρωση του τελικού χρόνου αναμονής μέσω της κάρτας διασύνδεσης δικτύου (NIC) και ο υπολογισμός της κατανομής χρόνου αναμονής.

Μονάδα αντιστοίχισης. Οι υπηρεσίες αντιστοιχίζονται στους πυρήνες χρησιμοποιώντας το Linux sched setaffinity system call και οι πυρήνες έχουν αντιστοιχιστεί στα cache ways με χρήση του Intel CAT.

Παράμετροι νευρωνικού δικτύου. Με πειραματική ανάλυση διαπιστώνουμε ότι οι ακόλουθες υπερ-παράμετροι απέδωσαν την καλύτερη κατανομή πόρων διατηρώντας την εγγύηση του QoS. Χρησιμοποιούμε τον Adam βελτιστοποίητη με learning rate 0,0025. Ορίζουμε το μέγεθος του minibatch σε 64 και το discount factor σε 0,99. Το δίκτυο προορισμού ενημερωνόταν κάθε 150 βήματα. Η προσάρτηση ξεκινά από το 1 και πέφτει στο 0,01. Χρησιμοποιήσαμε rectified non-linearity (ReLU) για όλα τα κρυφά επίπεδα και γραμμική ενεργοποίηση για τα επίπεδα εξόδου. Το δίκτυο έχει τρία κρυφά επίπεδα 512,256 και 128. Χρησιμοποιήσαμε prioritised experience replay με μέγεθος buffer 10^6 και $pr_\alpha = 0.6$ και γραμμική προσάρτηση $pr_\beta = 0.4$ έως 1 σε 10^{-8} βήματα και $\phi = 50$.

5 Πειραματική Αξιολόγηση

Για τους σκοπούς αυτής της μελέτης χρειαζόμασταν ένα εργαλείο συγκριτικής αξιολόγησης που θα προσέφερε τα περισσότερα από τις πιο δημοφιλείς εφαρμογές cloud και έχει τη δυνατότητα να αλλάζει τις αιτήσεις ανά δευτερόλεπτο δυναμικά κατά το χρόνο εκτέλεσης. Για να το πετύχουμε αυτό χρησιμοποιήσαμε την Tailbench με ορισμένες δευτερεύουσες προσαρμοσμένες τροποποιήσεις ώστε να διαβάζει τα RPS με την πάροδο του χρόνου από ένα αρχείο αφού ο πάγκος προσαρμογής λειτουργεί κανονικά με σταθερά RPS. Το αρχείο περιέχει RPS και χρόνους εκτέλεσης για το κάθε RPS. Για τους σκοπούς αυτής της μελέτης χρησιμοποιήσαμε το RPS που βρήκαμε στο πίνακα 4.3 ως το μέγιστο φορτίο κάθε εφαρμογής. Έχωντας αυτο υπόψιν, ρυθμίσαμε το RPS να αυξάνει αργά στο μέγιστο φορτίο και στη συνέχεια να μειώνεται ξανά έτσι ώστε το νευρικό δίκτυο να μπορεί να εκπαιδεύσει σε όλες τις καταστάσεις εφαρμογών.

5.1 Performance Monitoring Counters (PMCs)

Σε αυτο το κεφάλαιο θα προσδιορίσουμε την σχέση μεταξύ τον PMC και του RPS.

Pearson Correlation μεταξύ PMCs και RPS

Βλέποντας τα σχήματα 7 και 8 μπορεί να δεί κανείς ξεκάθαρα μια σχέση μεταξύ τον PMCs και του RPS. Υπολογίζοντας το Pearson Correlation μεταξυ τον PMCs

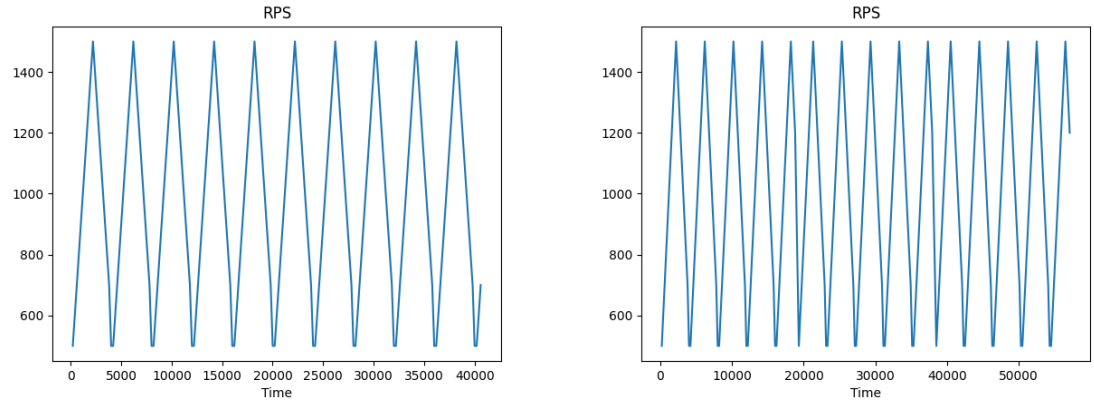

Figure 7: (a) img-dnn (b) masstree

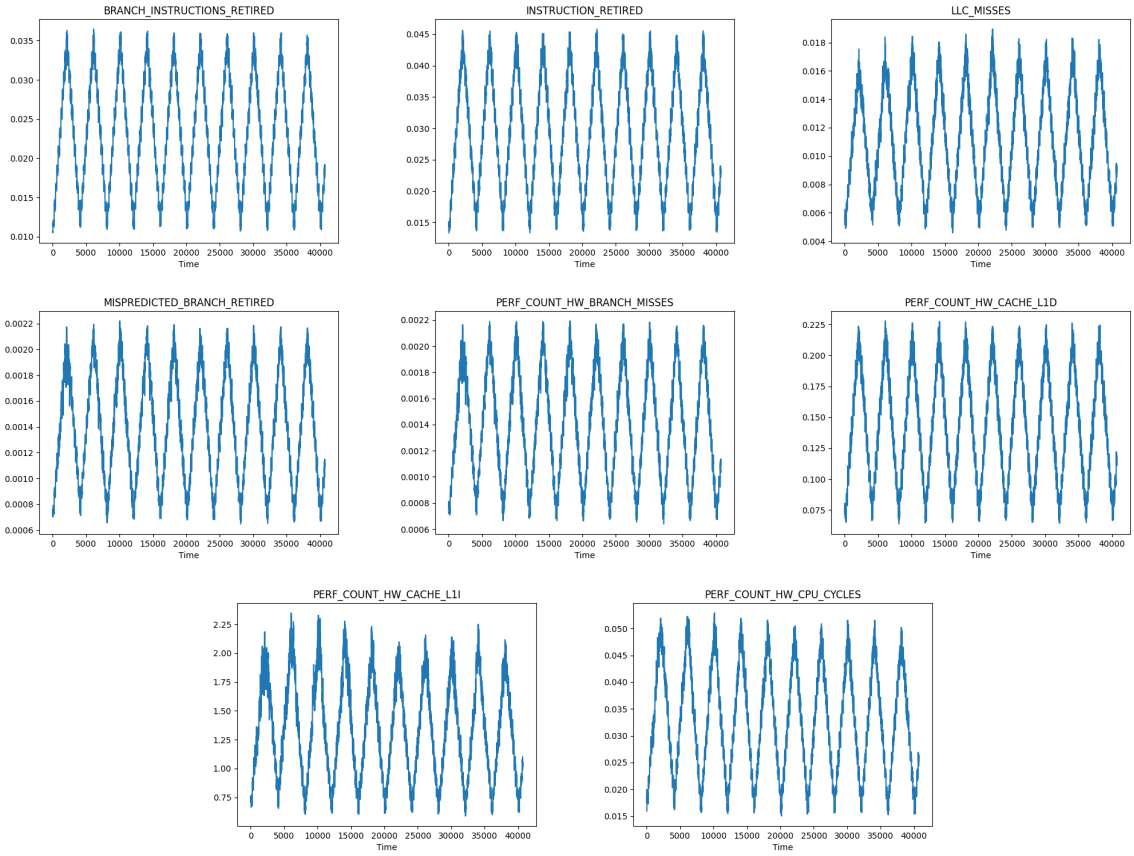

Figure 8: Performance Monitoring Counters

και του RPS βγάλαμε τον πίνακα 5.

Στη στατιστική, ο συντελεστής συσχέτισης Πεαρσον είναι ένα μέτρο γραμμικής συσχέτισης μεταξύ δύο συνόλων δεδομένων. Ο λόγος για τον οποίο θα θέλαμε να βρούμε τη συσχέτιση Πεαρσον μεταξύ των PMCs και του RPS είναι επειδή σε αυτή την ενότητα θα θέλαμε να αποδείξουμε ότι με την ύπαρξη των PMCs μπορεί κανείς να προβλέψει το τρέχον RPS (δεδομένου ότι σε ένα πραγματικό δυναμικό σύστημα μπορούμε να λάβουμε το RPS μόνο για το προηγούμενο χρονικό βήμα). Δείχνοντας ότι μπορεί κανείς να προβλέψει το τρέχον RPS από τα PMCs, δείχνουμε επίσης ότι το ίδιο το νευρωνικό δίκτυο μπορεί να συμπεράνει το RPS

από τα PMCs, οπότε το πρόβλημα διαχείρισης πόρων μεταφράζεται σε πρόβλημα μετατροπής του RPS σε δράση, το οποίο είναι πολύ απλούστερο.

Table 5: Pearson Correlation Between PMCs and RPS for img-dnn

PMC	Pearson Correlation
BRANCH INSTRUCTIONS RETIRED	0.99
INSTRUCTIONS RETIRED	0.99
LLC MISSES	0.99
MISSPREDICTED BRANCH RETIRED	0.99
PERF COUNT HW BRANCH MISSES	0.99
PERF COUNT HW CACHE L1D	0.99
PERF COUNT HW CACHE L1I	0.99
PERF COUNT HW CPU CYCLES	0.99
UNHALTED CORE CYCLES	0.99
UNHALTED REFERENCE CYCLES	0.99
UOPS RETIRED	0.99

5.2 Αξιολόγηση Επιδόσεων Χωρίς Παρεμβολές

img-dnn

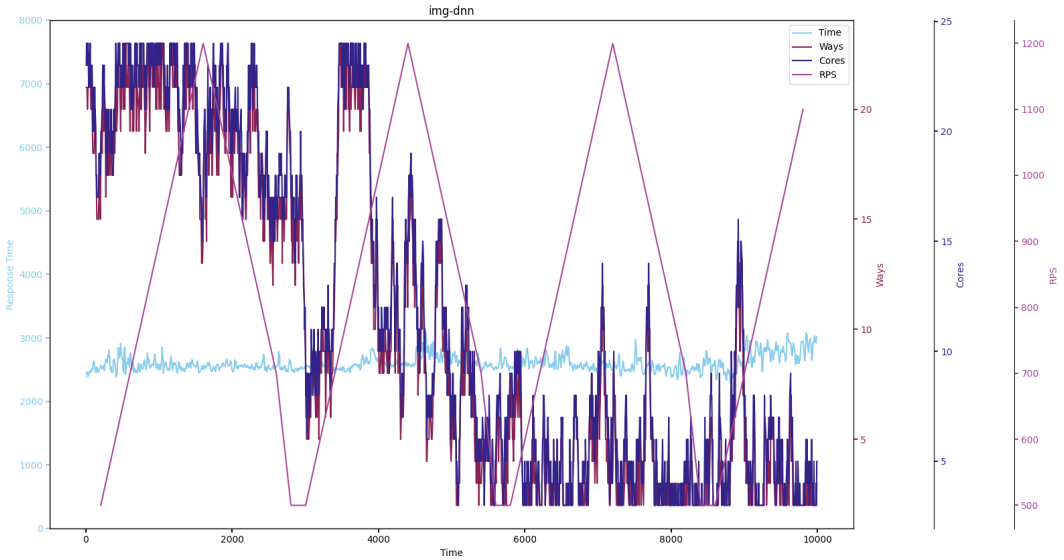

Figure 9: img-dnn during the first 10.000 seconds of training

Στο ανώτερο σχήμα μπορούμε να δούμε τα αποτελέσματα της εκπαίδευσης στα πρώτα 10 000 δευτερόλεπτα και της λειτουργίας αφότου εκπαιδευτηκέ του νευρωνικού δίκτυο στο img-dnn. Όπως μπορούμε να δούμε ο αριθμός των πυρήνων και τα cache ways καταλήγουν να ακολουθούν το RPS σε συμπεριφορά όπως το νευρωνικό δίκτυο εκπαιδεύεται. Ο χρόνος απόκρισης βρίσκεται σχεδόν πάντα

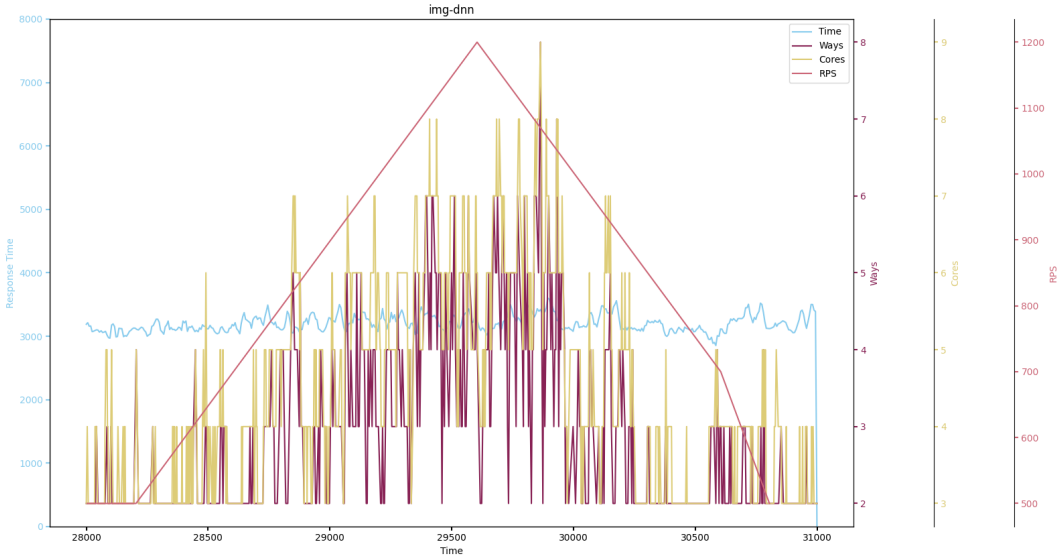

Figure 10: img-dnn once it has trained

υπό τον στόχο QoS ακόμη και στη χειρότερη περίπτωση εκπαίδευσης που είναι στα 8000 δευτερόλεπτα. Μπορούμε επίσης να δούμε ότι αξιοποιώντας μια καλή κατανομή πόρων ο ελεγκτής πετυχαίνει υψηλή διατήρηση QoS χρησιμοποιώντας μόνο 2 έως 8 πυρήνες και 2 έως 6 cache ways. Στο σχήμα παρακάτω μπορούμε επίσης να δούμε την εγγύηση του QoS όσο η εφαρμογή εκτελείται.

Κέρδος Πόρων.Κατά τη διάρκεια της παραπάνω εκτέλεσης img-dnn ο μέσος χρόνος απόκρισης για το εκπαιδευμένο νευρωνικό δίκτυο ήταν 1,12 χιλιοστά του δευτερολέπτου. Για να επιτύχουμε αυτό το είδος μέσου χρόνου απόκρισης, διατηρώντας παράλληλα μια εγγύηση QoS άνω του 99% για όλη τη διάρκεια ενός κύκλου, χωρίς τη χρήση του διαχειριστή πόρων και απλά θέτοντας την εφαρμογή σε σταθερό αριθμό πόρων, θα χρειαζόμασταν 8 πυρήνες CPU και 8 cache ways. Με τον υπολογισμό του μέσου όρου των πυρήνων CPU και των cache ways αποθήκευσης κατά τη διάρκεια ενός κύκλου στο σχήμα 6.8 πήραμε 5,23 πυρήνες CPU και 4,12 cache ways. Όπως βλέπουμε, έχουμε μείωση κατά 34% στη μέση χρήση της CPU και μείωση κατά 48% στη μέση χρήση cache ways.

masstree

Κέρδος Πόρων.Κατά τη διάρκεια της παραπάνω εκτέλεσης masstree, ο μέσος χρόνος απόκρισης για το εκπαιδευμένο νευρωνικό δίκτυο ήταν 1,12 χιλιοστά του δευτερολέπτου. Για να επιτύχουμε αυτό το είδος μέσου χρόνου απόκρισης, διατηρώντας παράλληλα μια εγγύηση QoS πάνω από 99% για όλη τη διάρκεια ενός κύκλου χωρίς τη χρήση του διαχειριστή πόρων και απλά θέτοντας την εφαρμογή με κρίσιμη καθυστέρηση σε σταθερό αριθμό πόρων, θα χρειαζόμασταν 8 πυρήνες CPU και 8 cache ways. Με τον υπολογισμό του μέσου όρου των πυρήνων CPU

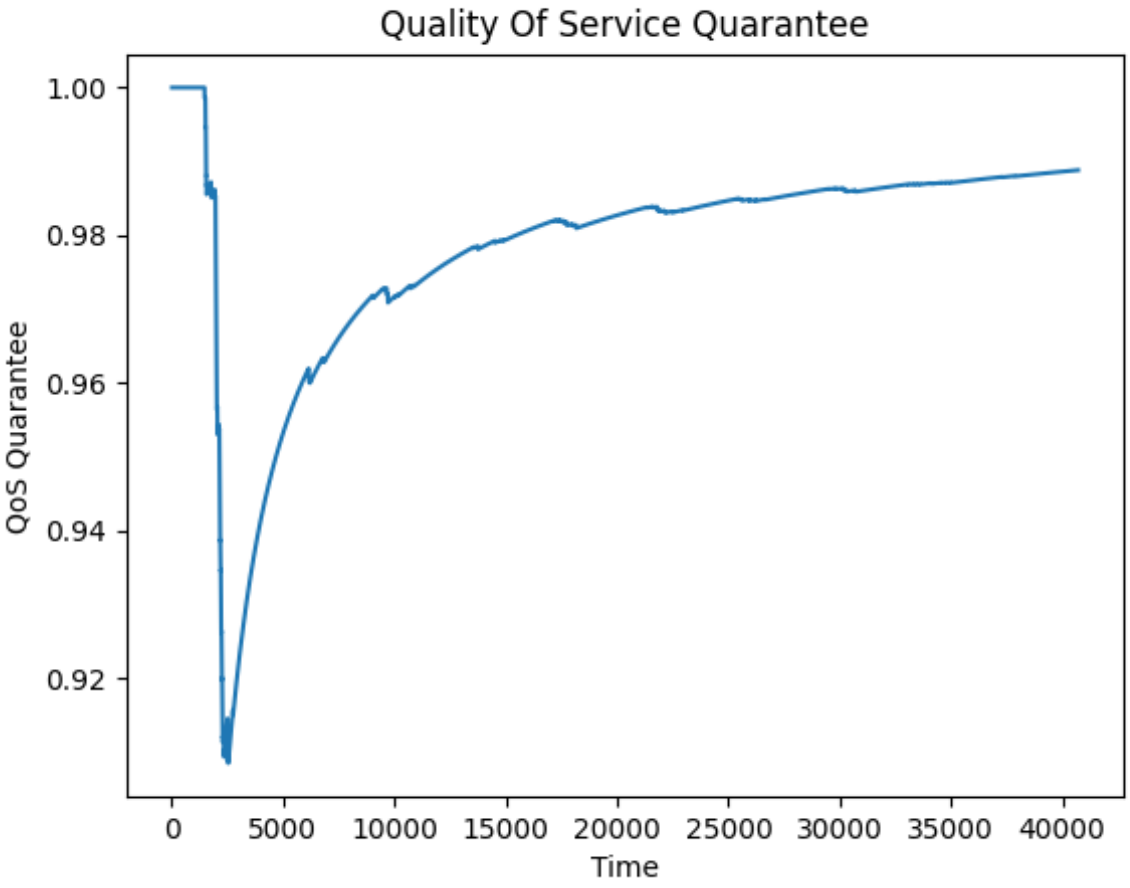

Figure 11: img-dnn QoS Quarantee

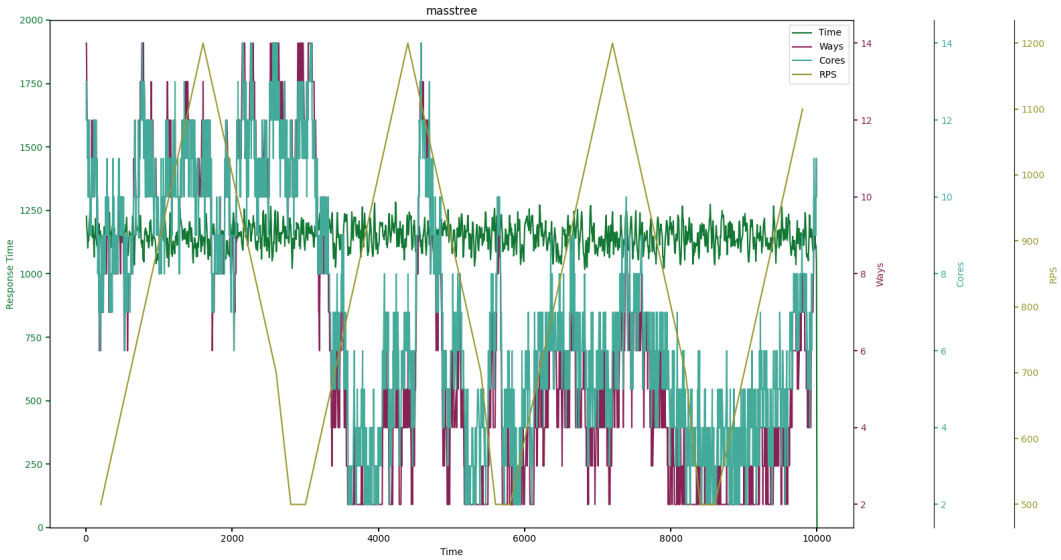

Figure 12: Masstree during the first 10.000 seconds of training

και των cache ways κατά τη διάρκεια ενός κύκλου στο σχήμα 6.8 πήραμε 5,23 πυρήνες CPU και 4,12 cache ways. Όπως βλέπουμε, έχουμε μείωση κατά 34%

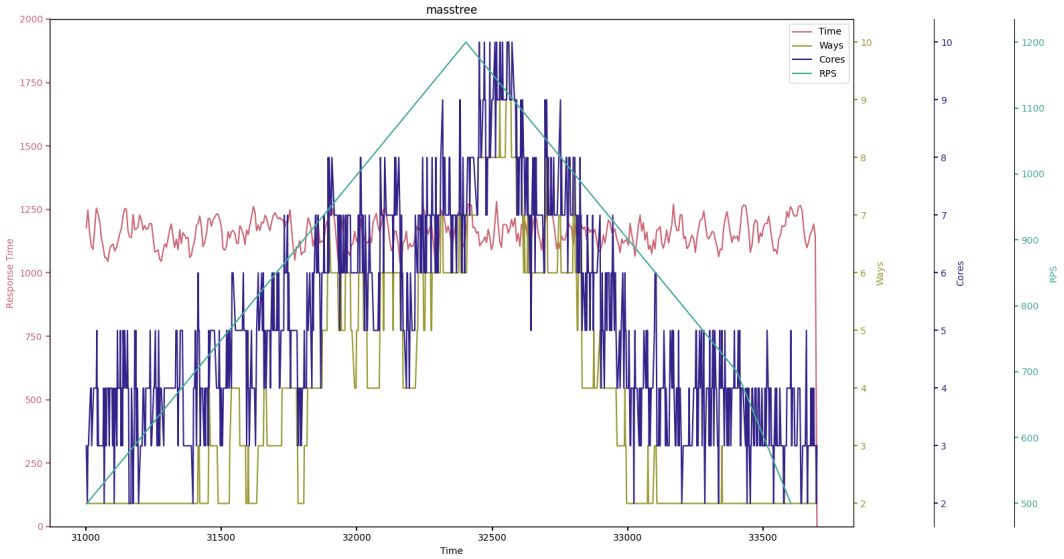

Figure 13: Masstree once it has trained

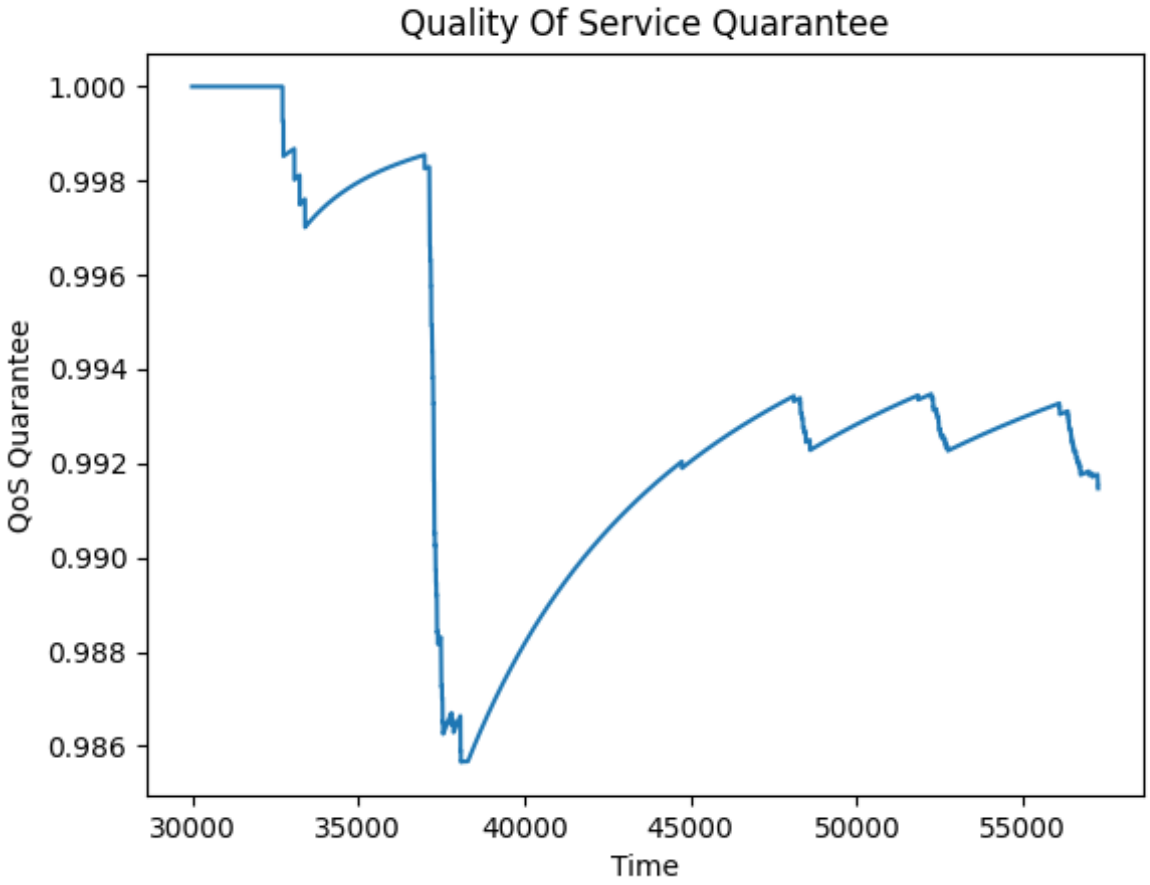

Figure 14: img-dnn QoS Quarantee on a trained model

στη μέση χρήση της CPU και μείωση κατά 48% στη μέση χρήση των cache ways.

5.3 Αξιολόγηση Επιδόσεων Υπό Συνθήκες Παρεμβολής

Ταυτόχρονη Εκτέλεση Εφαρμογών LC

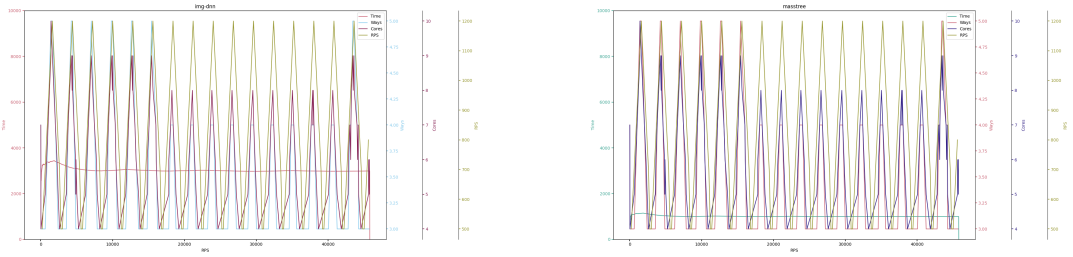

Figure 15: (a) img-dnn (b) masstree

Στα σχήματα 6.12 είχαμε ένα εκπαιδευμένο μοντέλο img-dnn και ένα εκπαιδευμένο masstree μοντέλο να εκτελούνται παράλληλα χωρίς παρεμβολές. Όπως βλέπουμε το νευρωνικό δίκτυο επιτυγχάνει σταθερό χρόνο απόκρισης εντός του στόχου του QoS. Μπορούμε επίσης να δούμε πώς η συνολική χρήση πόρων είναι πολύ μικρότερη από 100%.

Ταυτόχρονη Εκτέλεση Εφαρμογών LC και Batch Workloads

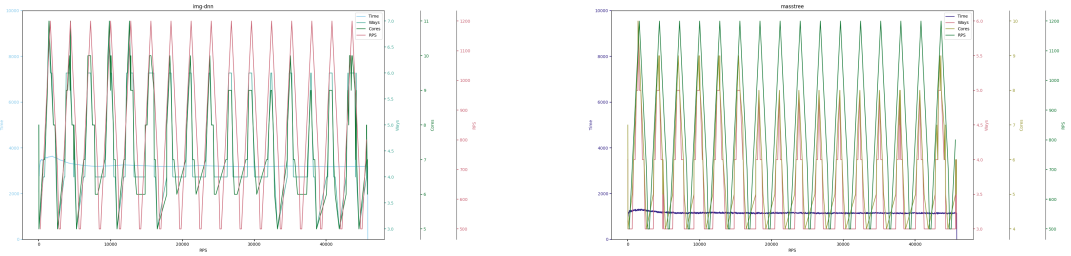

Figure 16: (a) img-dnn (b) masstree

Στα σχήματα 6.13 είχαμε ένα εκπαιδευμένο μοντέλο img-dnn και ένα εκπαιδευμένο μοντέλο masstree να εκτελούνται παράλληλα με παρεμβολή. Η παρέμβαση σε αυτή τη μελέτη είναι απλά batch workloads. Όπως βλέπουμε το νευρωνικό δίκτυο επιτυγχάνει σταθερή ανταπόκριση που είναι εντός του στόχου QoS. Μπορούμε επίσης να δούμε πώς η συνολική χρήση των πόρων είναι πολύ μικρότερη από 100%. Για παράδειγμα στις κορυφές το μέγιστο πλήθος απο πυρήνες που χρησημοποιούνται απο το το img-dnn είναι 9-10 και οι μέγιστοι πυρήνες που χρησιμοποιούνται από το masstree είναι 8. Ωστόσο το σύστημα έχει συνολικά 24 πυρήνες, βλέπουμε ότι ακόμη και στην κορυφή του RPS εξακολουθούν να απομένουν 6 πυρήνες για batch workloads. Με την ίδια λογική ο αριθμός τον cache ways που χρησιμοποιούνται κατά τη διάρκεια των κορυφών είναι 6 για img-dnn και 5 για masstree ενώ το σύστημα έχει 20 cache ways. Για τον σκοπό αυτής της εργασίας το batch workload ήταν το sphinx benchmark του Tailbench.

5.4 Επιβάρυνση Του Ελεγκτή Κατά Την Εκτέλεση Εφαρμογών

Ο υπολογισμός του overhead έγινε με δύο διαφορετικές μεθοδολογίες.

Για την πρώτη μεθοδολογία θα τρέξουμε το benchmark για ενα σταθερό πλήθος απο αιτήσεις με και χωρίς τον διαχειριστή πόρων.

Για την δεύτερη μεθοδολογία θα υπολογίσουμε πόσο χρόνο παίρνει κάθε στάδιο του διαχειριστή πόρων στο run-time.

Οσο αφορά την πρώτη μεθοδολογία θα υπολογίσουμε το overhead για τρεις διαφορετικές περιπτώσεις. Η μία περίπτωση είναι να έχουμε τον διαχειριστή πόρων και την εφαρμογή σε διαφορετικά sockets αλλά στο ίδιο σύστημα. Η δεύτερη περίπτωση είναι να έχουμε την εφαρμογή και το διαχειριστή πόρων στο ίδιο socket αλλα σε διαφορετικούς physical cores. Και στην τελευταία περίπτωση θα έχουμε την εφαρμογή και το διαχειριστή πόρων στο ίδιο physical core.

Overhead Calculation Using Fixed Requests

Τρέξαμε το benchmark για 100.000 αιτήσεις χωρίς τον διαχειριστή πόρων και για 100.000 αιτήσες με τον διαχειριστή πόρων. Αυτό το κάναμε 100 φορές και πήραμε τους μέσους όρους.

	Different Sockets	**Different Physical Cores**	**Same Physical Core**
With Neural Network	392	398	404
Without Neural Network	392	392	392
Overhead	0%	1.6%	3%

Table 6: Χρόνος εκτέλεσης σε δευτερόλεπτα

Overhead Calculation Using Run-time

Η δεύτερη μεθοδολογία αφορά τον υπολογισμό χρόνου για κάθε κομμάτι του διαχειριστή πόρων. Ο διαχειριστής πόρων τρέχει κάθε δύο δευτερόλεπτα άρα το overhead θα ήταν ο παραπάνω χρόνος.

Table 7: Title

Gradient descent computation	21 ms
Gather and preprocess PMCs	3 ms
Core/Way Allocation	8 ms
Total Overhead	32 ms

$$Overhead = \frac{2.032 - 2}{2} = 0.015 = 1.5\% \tag{2}$$

Όπως βλέπουμε απο την πρώτη μεθοδολογία βρήκαμε overhead 1.6% ενώ από την δεύτερη μέθοδο βρήκαμε overhead 1.5% στην περίπτωση που η εφαρμογή και

ο διαχειριστής πόρων ειναι στο ίδιο socket αλλά σε διαφορετικά physical cores, άρα μπορούμε να πούμε οτι οι δύο μέθοδοι συμφωνούνε στα αποτελέσματα.

Chapter 1

Introduction

1.1 Cloud Computing

Cloud computing has two meanings. The most common refers to running workloads remotely over the internet in a commercial provider's data center, also known as the "public cloud" model. Popular public cloud offerings—such as Amazon Web Services (AWS), Salesforce's CRM system, and Microsoft Azure—all exemplify this familiar notion of cloud computing. Today, most businesses take a multi-cloud approach, which simply means they use more than one public cloud service.

The second meaning of cloud computing describes how it works: a virtualized pool of resources, from raw compute power to application functionality, available on demand. When customers procure cloud services, the provider fulfills those requests using advanced automation rather than manual provisioning. The key advantage is agility: the ability to apply abstracted compute, storage, and network resources to workloads as needed and tap into an abundance of prebuilt services.

The public cloud lets customers gain new capabilities without investing in new hardware or software. Instead, they pay their cloud provider a subscription fee or pay for only the resources they use. Simply by filling in web forms, users can set up accounts and spin up virtual machines or provision new applications. More users or computing resources can be added on the fly—the latter in real time as workloads demand those resources thanks to a feature known as autoscaling.

The array of available cloud computing services is vast, but most fall into either Infrastracture as a Service (IaaS), Platform as a Service (PaaS) or Software as a Service (SaaS).

At a basic level, IaaS public cloud providers offer storage and compute services on a pay-per-use basis. But the full array of services offered by all major public cloud providers is staggering: highly scalable databases, virtual private networks, big data analytics, developer tools, machine learning, application

monitoring, and so on. Amazon Web Services was the first IaaS provider and remains the leader, followed by Microsoft Azure, Google Cloud Platform, and IBM Cloud.

PaaS provides sets of services and workflows that specifically target developers, who can use shared tools, processes, and APIs to accelerate the development, testing, and deployment of applications. Salesforce's Heroku and Force.com are popular public cloud PaaS offerings; Pivotal's Cloud Foundry and Red Hat's OpenShift can be deployed on premises or accessed through the major public clouds. For enterprises, PaaS can ensure that developers have ready access to resources, follow certain processes, and use only a specific array of services, while operators maintain the underlying infrastructure.

Lastly SaaS delivers applications over the internet through the browser. The most popular SaaS applications for business can be found in Google's G Suite and Microsoft's Office 365; among enterprise applications, Salesforce leads the pack. But virtually all enterprise applications, including ERP suites from Oracle and SAP, have adopted the SaaS model. Typically, SaaS applications offer extensive configuration options as well as development environments that enable customers to code their own modifications and additions.

1.1.1 Cloud Resource Usage Optimization

Even though one of the reasons for using a public cloud provider is the flexibility of auto-scaling your resources to match your needs most organizations buy their resources in advance and in bulk so as to be able to get a substantial discount on the cost of the aforementioned resources. This very phenomena creates the need for organizations to be able to optimally use their resources so as to not waste money. Many organizations host latency critical application using their public cloud providers but this application are prone to workload instabilities during the passing of day because human interaction with the aforementioned applications depends on the human behaviour throughout the day. As a result the actual resources needed for an organization varies during the passing of a day or even a weekend or month. What many organization do to maximize resource utilization is run batch workloads alongside their latency critical applications. A batch workload is defined as a long-running best-effort workload with no Quality of Service (QoS) restraints. Even though batch workloads are long-running and best-effort they need to be ran as quickly as possible otherwise you may end up with an ever increase queue of batch workloads, but in the same time a batch workload can create interference for the latency critical application and as a result deny the QoS guarantee of the latency critical application.

So with everything mentioned above when thinking of Cloud Resource Usage Optimization one ends up having to balance many variables so as to satisfy

many different targets.

1.2 Artificial Intelligence

Artificial intelligence (AI), the ability of a digital computer or computer-controlled robot to perform tasks commonly associated with intelligent beings. The term is frequently applied to the project of developing systems endowed with the intellectual processes characteristic of humans, such as the ability to reason, discover meaning, generalize, or learn from past experience. Since the development of the digital computer in the 1940s, it has been demonstrated that computers can be programmed to carry out very complex tasks—as, for example, discovering proofs for mathematical theorems or playing chess—with great proficiency. Still, despite continuing advances in computer processing speed and memory capacity, there are as yet no programs that can match human flexibility over wider domains or in tasks requiring much everyday knowledge. On the other hand, some programs have attained the performance levels of human experts and professionals in performing certain specific tasks, so that artificial intelligence in this limited sense is found in applications as diverse as medical diagnosis, computer search engines, and voice or handwriting recognition[2].

1.2.1 What is Intelligence?

All but the simplest human behaviour is ascribed to intelligence, while even the most complicated insect behaviour is never taken as an indication of intelligence. What is the difference? Consider the behaviour of the digger wasp, Sphex ichneumoneus. When the female wasp returns to her burrow with food, she first deposits it on the threshold, checks for intruders inside her burrow, and only then, if the coast is clear, carries her food inside. The real nature of the wasp's instinctual behaviour is revealed if the food is moved a few inches away from the entrance to her burrow while she is inside: on emerging, she will repeat the whole procedure as often as the food is displaced. Intelligence—conspicuously absent in the case of Sphex—must include the ability to adapt to new circumstances.

Psychologists generally do not characterize human intelligence by just one trait but by the combination of many diverse abilities. Research in AI has focused chiefly on the following components of intelligence: learning, reasoning, problem solving, perception, and using language.

1.2.2 Methods and goals in AI

AI research follows two distinct, and to some extent competing, methods, the symbolic (or "top-down") approach, and the connectionist (or "bottom-up") approach. The top-down approach seeks to replicate intelligence by analyzing cognition independent of the biological structure of the brain, in terms of the processing of symbols—whence the symbolic label. The bottom-up approach, on the other hand, involves creating artificial neural networks in imitation of the brain's structure—whence the connectionist label.

1.2.3 Strong AI, applied AI, and cognitive simulation

AI research attempts to reach one of three goals: strong AI, applied AI, or cognitive simulation. Strong AI aims to build machines that think. (The term strong AI was introduced for this category of research in 1980 by the philosopher John Searle of the University of California at Berkeley.) The ultimate ambition of strong AI is to produce a machine whose overall intellectual ability is indistinguishable from that of a human being. As is described in the section Early milestones in AI, this goal generated great interest in the 1950s and '60s, but such optimism has given way to an appreciation of the extreme difficulties involved. To date, progress has been meagre. Some critics doubt whether research will produce even a system with the overall intellectual ability of an ant in the foreseeable future. Indeed, some researchers working in AI's other two branches view strong AI as not worth pursuing.

Applied AI, also known as advanced information processing, aims to produce commercially viable "smart" systems—for example, "expert" medical diagnosis systems and stock-trading systems. Applied AI has enjoyed considerable success, as described in the section Expert systems.

In cognitive simulation, computers are used to test theories about how the human mind works—for example, theories about how people recognize faces or recall memories. Cognitive simulation is already a powerful tool in both neuroscience and cognitive psychology.

1.3 CPU Cache Memory

Cache memory, also called cache, supplementary memory system that temporarily stores frequently used instructions and data for quicker processing by the central processing unit (CPU) of a computer. The cache augments, and is an extension of, a computer's main memory. Both main memory and cache are internal random-access memories (RAMs) that use semiconductor-based transistor circuits. Cache holds a copy of only the most frequently used information

or program codes stored in the main memory. The smaller capacity of the cache reduces the time required to locate data within it and provide it to the CPU for processing.

When a computer's CPU accesses its internal memory, it first checks to see if the information it needs is stored in the cache. If it is, the cache returns the data to the CPU. If the information is not in the cache, the CPU retrieves it from the main memory. Disk cache memory operates similarly, but the cache is used to hold data that have recently been written on, or retrieved from, a magnetic disk or other external storage device[3].

1.4 Multithreaded CPUs

Multicore and multithreaded CPUs have become the new approach to obtaining increases in CPU performance. Multithreaded and multicore CPUs both exploit concurrency by executing multiple threads, although their designs target different objectives. Multicore CPUs achieve thread concurrency at a higher level, focusing less on utilization per core and aiming at scalability through replicating cores. These CPUs are often called chip multiprocessors (CMPs). All multithreaded cores keep multiple hardware threads on-chip and ready for execution. Each on-chip thread needs its own state components, such as the instruction pointer and other control registers. Aside from concurrency, caches are the most important feature for enhancing modern CPU performance because of the gap between CPU speed and memory-access times. Another important feature that impacts multicore chip performance is the communication among different on-chip components: cores, caches, and memory controllers and network controllers[4].

1.5 Thesis Overview

The presented work employs the usage of modern deep reinforcement learning algorithms to provide an alternative resource manager that is able to overcome the challenges of latency critical application co-location in the modern cloud environment and maximize the resource usage while still maintaining an acceptable level of performance.

While many different approaches have been developed to tackle this challenge, this thesis offers a novel approach with the following contributions:

- We present the characteristics of a modern cloud service. We take in account the issues of the current cloud environment and we present the importance of resources in modern cloud services.

- We describe, design and implement a resource manager that uses deep reinforcement learning so as to resource allocation actions that maximize the resource usage while maintaining an acceptable level of performance.
- We also demonstrate how our resource manager works with managing multiple applications while also running batch workloads.
- We evaluate our resource manager and compare it to the classical way of managing resources in cloud services using a custom version of Tailbench[1]. We also present the resource usage gain and the overhead of our resource manager.

The present thesis is organized as follows:

- In chapter 2, we summarize existing work related to our research.
- In chapter 3, we give a brief introduction in modern Machine Learning and specifically we focus on Deep Reinforcement Learning as well as an introduction to Intel's Cache Allocation Technology and taskset for core partitioning.
- In chapter 4, we study some of the most well known single core benchmarks and some of the most well known multi threaded cloud benchmarks so as to be able to characterize each application.
- In chapter 5, we describe the resource manager we developer for this work.
- In chapter 6, we assess the resource manager, we describe how we implemented it and it's tests, test it and analyze the results of our experiments.
- Finally, in chapter 7, we draw conclusions from our work and provide ideas for future research.

Chapter 2

Related Work

In this chapter we will present some work that approaches the same problems or related problems. Most of the related work tries to solve the problem with classical non deep reinforcement learning based methods. There is also one related work that solves this problem using deep reinforcement learning but it has a different approach than ours.

2.1 Our Approach

Our approach is based on the assumption that small fine-grained changes are best for the system, so we use deep reinforcement learning together with small fine-grained actions in our resource manager to achieve our goal and minimize the overhead of our work. Also none of the related work tries to solve the cache partitioning problem using deep reinforcement learning or any other machine learning based technique.

2.2 Dynamic Resource Management on Cloud Infrastructures

2.2.1 PARTIES

PARTIES[5] is a QoS-aware resource manager that enables an arbitrary number of interactive, latency-critical services to share a physical node without QoS violations.PARTIES leverages a set of hardware and software resource partitioning mechanisms to adjust allocations dynamically at runtime, in a way that meets the QoS requirements of each co-scheduled workload, and maximizes throughput for the machine. Results show that PARTIES improves throughput under QoS by 61 percent on average, compared to existing resource managers,and that the rate of improvement increases with the number of co-scheduled applications per physical host.

2.2.2 KPart

KPart[6] is a hybrid cache partitioning-sharing technique that sidesteps the limitations of way-partitioning and unlocks significant performance on current systems. KPart first groups applications into clusters, then partitions the cache among these clusters. To build clusters, KPart relies on a novel technique to estimate the performance loss an application suffers when sharing a partition. KPart automatically chooses the number of clusters, balancing the isolation benefits of way- partitioning with its potential performance impact. KPart uses detailed profiling information to make these decisions. This information can be gathered either offline, or online at low overhead using a novel profiling mechanism.

2.2.3 CoPart

CoPart[7] is coordinated partitioning of LLC and memory bandwidth for fairness-aware workload consolidation on commodity servers. CoPart dynamically analyzes the characteristics of the consolidated applications and allocates the LLC and memory bandwidth across the applications in a coordinated manner to improve the overall fairness. Our quantitative evaluation shows that CoPart significantly improves the fairness of the consolidated applications, robustly provides high fairness across various application and system configurations, and incurs small performance overhead.

2.2.4 Bubble-Flux

Bubble-Flux[8] is an integrated dynamic interference measurement and online QoS management mechanism that provides accurate QoS control and maximizes server utilization. It is consisted of two parts. The first one, Dynamic Bubble, measures the instantaneous pressure on the shared hardware resources and predict how the QoS of a latency-sensitive job will be affected by potential co-runners. Secondly, using an online Bubble Flux Engine, monitors the QoS of the latency-sensitive applications and controls the execution of batch jobs to adapt to load changes, in order to deliver satisfactory QoS.

2.3 Deep Reinforcement Learning Approaches on System's Research

2.3.1 TWIG

Twig[9] is a scalable quality-of-service (QoS) aware task manager for latency-critical services co-located on a server system. Twig successfully leverages deep

reinforcement learning to characterise tail latency using hardware performance counters and to drive energy-efficient task management decisions in data centres. Results show that Twig outperforms prior works in reducing energy usage by up to 38 percent while achieving up to 99 percent QoS guarantee for latency-critical services.

Chapter 3

Background on Deep Reinforcement Learning and Allocation Technologies

3.1 What is Machine Learning?

Machine learning is a broad term that refers to automated computer methods that learn a task from a set of examples and are based on logical or binary operations. The decision-tree technique, in which categorization is achieved by a series of logical processes, has received a lot of attention. Given enough data, they are capable of expressing even the most complicated problems.

Other techniques, such as genetic algorithms and inductive logic procedures, are currently in development and, in theory, would allow us to deal with a wider range of data, including cases where the number and type of attributes vary, where additional layers of learning are superimposed, and where attributes and classes are organized hierarchically, and so on. Machine Learning tries to provide classification statements that are basic enough for humans to understand. They must be able to sufficiently replicate human reasoning in order to give insight into the decision-making process.

Background knowledge, including statistical methods, can be used in development, but operation is supposed to be without human involvement. To learn is to gain knowledge, comprehension, or mastery of anything by experience or study, or to gain knowledge (of something) or develop competence in (some art or activity), to get experience of or an ability or a skill in, to memorize (something). Machine learning may be defined as the act of creating computer systems that develop automatically over time and incorporate a learning process.

Machine learning is still described as learning a theory automatically from data by a process of inference, model fitting, or learning from instances such as these. In the absence of a general theory, automated extraction of meaningful information from a body of data by developing excellent probabilistic models is best suited for fields with plenty of data [10].

3.2 Deep Reinforcement Learning

Deep learning, a type of machine learning that has acquired a lot of popularity in recent years because to its incredible outcomes in a variety of applications like pattern identification, audio recognition, computer vision, and natural language processing. Deep learning approaches may also be coupled with reinforcement learning methods to develop meaningful representations for issues with high dimensional raw data input, according to recent study[11].

3.2.1 Reinforcement Learning Algorithms

Reinforcement learning (RL) algorithms use a technique for learning by interacting with the environment (via a series of actions, observations, and rewards). In a range of problems ranging from robotics to resource allocation, RL-based techniques have demonstrated significant effectiveness. As a result, they've emerged as one of the most promising prospects for achieving artificial intelligence's (AI) aim of creating autonomous agents that can learn in complicated and unpredictable settings.

A good representation of the environment in which the agent is to be learnt is required before an agent or robot (software or hardware) may choose an action. As a result, perception is one of the major issues that must be resolved before the agent can choose the best course of action.

The environment might be supplied or obtained as a representation. In reinforcement learning tasks, a human expert generally offers hand-crafted environment characteristics based on his task understanding. This approach, on the other hand, is too complex, if not impossible, for some real-world control issues involving high-dimensional sensory input such as vision and voice. Furthermore, the quality of the feature representation has a significant impact on the performance of such learning. Furthermore, generalizing a particular approach to various problems may not be successful. As a result, it should be done automatically, because the precision that automatic feature extraction can give is far higher, and the method will not be affected by difficulties with hand-designed features.

Deep learning research has proven in recent years that it is a highly promising and powerful technique for automatically extracting features from raw data, such as raw pixels in a picture. It has attracted a lot of attention not only from academics (due to its performance in a variety of applications like pattern recognition, speech recognition, computer vision, and natural language processing), but also from tech giants like Google (Google Translate, Image search engine), Apple (Apple's Siri), Microsoft (Bing voice search), and others.

Some supervised and unsupervised deep learning approaches, such as mul-

tilayer perceptrons (MLPs), convolutional neural networks (CNNs), auto encoders, and recurrent neural networks (RNNs), have recently begun to include reinforcement learning methodologies. Deep learning approaches may also be used to develop useful representations for reinforcement learning issues, according to the evaluation of the new algorithms and methods that resulted. By combining RL with deep learning approaches, an RL agent may get a strong understanding of its surroundings by leveraging the capabilities of deep neural networks.

3.2.2 Reinforcement Learning

Reinforcement Learning is a type of machine learning in which a computer learns by interacting with its surroundings. An agent can learn through trial and error using an RL framework. The objective of the RL agent is to learn to pick behaviors that maximize the predicted cumulative reward over time by engaging in the environment. In other words, the agent seeks to learn an ideal sequence of activities to perform in order to achieve its objective by monitoring the outcomes of the actions it is doing in the environment.

A Markov decision process can be used to simulate a reinforcement learning agent (MDP). The issue is called a finite MDP if the states and action spaces are finite. Finite MDPs are critical for RL issues, and much of the literature assumes that the environment is a finite MDP.

In the finite MDP framework, an RL agent behaves as follows: The learning agent interacts with its surroundings by performing actions and getting feedback. The agent chooses an action a from a set of lawful actions $A = 1, 2, ..., k$ at state $s_t \in S$, where S is the set of potential states, at each time step t, which spans a collection of discrete time intervals. A policy is used to determine which actions should be taken. The policy is a description of the agent's behavior that instructs the agent on the actions to do in each conceivable condition. The agent obtains a scalar reward $r_t \in R$ for each action and observes the following state $s_{t+1} \in S$ one step time later.

A transition distribution $P(s_{t+1}|s_t, a_t)$ provides the probability of each potential next state s_{t+1}, where $s_{t+1}, s_t \in S, a_t \in A(s_t)$. Similar to the probability of each potential reward r_t, the probability of each possible reward r_t comes from a reward distribution $P(r_t|s_t, a_t)$, where $s_t \in S, a_t \in A(s_t)$. As a result, $E_{P(r_t|s_t,a_t)}(r_t|s_t = s, a_t = a)$ is used to compute the expected scalar reward received,r_t, by executing action a in current states.

The learning agent's goal is to learn an optimum policy π, which determines the probability of picking action a in states, so that the total of discounted rewards over time is maximized by following the policy. At time t, the expected discounted return R is defined as follows:

$$R_t = E[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ...] = E[\sum_{k=0}^{\infty} \gamma^{\kappa} r_{t+\kappa}] \tag{3.1}$$

The discount factor is $0 < \gamma < 1$, and $E[.]$ represents the expectation with regard to the reward distribution. The action-value function $Q_\pi(s, a)$ is defined as follows in terms of transition probabilities and anticipated discounted immediate rewards, which are crucial factors for defining dynamics of a finite MDP:

$$Q_\pi(s, a) = E_\pi[R_t|s_t = s, a_t = a] = E_\pi[\sum_{k=0}^{\infty} \gamma^{\kappa} r_{t+\kappa}|s_t = s, a_t = a] \tag{3.2}$$

The expected return achieved by beginning with states,$s \in S$, executing action a,$a \in A$, and then following policy pi, where pi is a mapping from states to actions or distributions over actions, is the action-value function $A_\pi(s, a)$ for an agent.

It is evident from the unfolding of equation 2 that it has a recursive feature, allowing the following iterative update to be employed for the estimate of the action-value function:

$$\begin{aligned} Q^{\pi}_{i+1}(s, a) &= E_\pi[r_t + \gamma \sum_{k=0}^{\infty} \gamma^{\kappa} r_{t+\kappa+1}|s_t = s, a_t = a] \\ &= E_\pi[r_t + \gamma Q^{\pi}_i(s_{t+1} = s^{'}, a_{t+1} = a^{'})|s_t = s, a_t = a] \end{aligned} \tag{3.3}$$

In Eq. 3.3, both states a link between the value of an action in a state and the values of its following actions that can be done for all s,$s^{'} \in S$ and a,$a^{'} \in A$. It also mentions a method for calculating the value depending on the previous ones. The goal of a reinforcement learning agent is to find a strategy that achieves the largest future reward during execution. As a result, it must learn an optimal policy π, a policy with an expected value greater than or equal to the anticipated value of following other policies for all states, and as a result, an optimal state-value function $Q(s, a)$.

$$Q_{i+1}(s, a) = E_\pi[r_t + \gamma max_{a'} Q_i(s^{'}, a^{'})|s, a] \tag{3.4}$$

Where it is assumed that $s, s \in S$ and $a, a \in A$ are equal. The value iteration method converges to the ideal action-value function, Q^*, as $i \to \infty$ after each iteration.

Many real-world issues have a high number of states and actions, making the traditional solution (a state-action table to record the values of state-action pairings) unworkable. The employment of a function approximator as an estimate of the action-value function is one solution. The parameter vector θ is used to parameterize the approximate value function $Q(s, a; \theta)$. Typically, gradient-descent techniques are used to learn parameters by attempting to minimize the mean-squared error in Q-values using the following loss function:

$$L(\theta) = E_{\pi}[(r + \gamma max_{a'} Q(s', a'; \theta) - Q(s, a; \theta))^2] \tag{3.5}$$

Where the desired value is $r + \gamma max_{a'} Q(s', a'; \theta)$. The gradient obtained by differentiating the loss function with regard to its parameters *theta* is as follows:

$$\frac{\partial L}{\partial \theta} = E_{\pi}[(r + \gamma max_{a'} Q(s', a'; \theta) - Q(s, a; \theta))^2] \frac{\partial Q(s, a; \theta)}{\partial \theta} \tag{3.6}$$

The gradient-based techniques are used in this manner. Typically, the stochastic gradient descent method is used to optimize the gradient above. The approximation function might be a linear or non-linear function of the parameters θ (for example, a neural network). The majority of work in reinforcement learning until recently relied on linear function approximators because of the convergence assurances they give. Not only traditional neural networks, such as multi-layer perceptrons (MLP), but also deep neural networks, such as convolutional neural networks and recurrent neural networks, have been widely used as function approximators for large reinforcement learning tasks in recent years to solve convergence problems.

3.2.3 Deep Learning

The ability of a machine learning approach to perform well is strongly dependent on the quality of its input data representation. As a result, data pre-processing (i.e., feature learning) is an important stage in the process of building robots that learn via observation of data. The feature engineering technique is a method of utilizing domain specialists' expertise in order to extract hand-crafted features and minimize the dimension of input data features.

Feature learning is required for the effectiveness of shallow learning models such as support vector machines (SVMs) and logistic regression. This is an important procedure, but it takes a long time and is tough to complete. It would be preferable to have algorithms that make the problem easier to solve. Deep learning techniques are one of the most effective methods for dealing with large amounts of data and extracting discriminative information. Feature extraction (the extraction of representations) from data may be automated using deep learning algorithms.

Without human expertise, the representation is leant by data that is put straight into deep networks (i.e., automated feature extraction). This important feature of deep learning architectures has aided progress toward the algorithms that are the objective of Artificial Intelligence (AI), allowing it to understand the world around it without the need for expert knowledge or meddling.

In order to learn from many layers of abstractions, deep learning aims to represent high-level abstractions in data using deep networks of supervised and/or unsupervised learning algorithms. It uses deep architectures to learn hierarchical representations for a variety of tasks, including classification.

Multiple layers of representations are used in deep learning models. It is, in fact, a collection of components including autoencoders, Restricted Boltzmann Machines (RBMs), and convolutional layers. The raw data is put into a network with many layers during training.

The outputs of each layer, which are nonlinear feature transformations, are utilized as inputs to the deep network's subsequent levels. The last layer's output representation can be utilized for limiting classifiers or applications that benefit from abstract representations of data in a hierarchical way as inputs. Each layer attempts to learn and uncover underlying explanatory elements by applying a nonlinear transformation to its input. As a result, this process leads to the formation of a hierarchy of abstract representations.

In image processing applications, for example, the first layer is given by the picture pixels, which might lead to learning the edges of distinct objects in the image. The second layer leans sophisticated characteristics such as object components using the representations supplied by the previous layer (combination of edges). To figure out object models, the third layer assembles object pieces (more complicated characteristics). The example demonstrates how a deep learning system may detect items in a picture using the hierarchical learning capability of abstracted representations. As a result, the deep learning method may be thought of as a form of representation learning algorithm.

Deep learning approaches may now be used to solve real-world problems thanks to the usage of deep neural networks. However, learning the parameters in a deep architecture is a challenging optimization process with a high computational complexity (deep networks with multiple hidden layers have millions of parameters to learn). Fortunately, the emergence of sophisticated parallel processing technology such as GPU has addressed this difficulty to some extent.

3.3 Cache Allocation Technology

3.3.1 Why is Cache Allocation Technology (CAT) needed?

Maintaining constant performance and prioritizing critical interactive apps may be difficult in today's data-center cloud environment, where multi-tenant VMs are common and typically run numerous diverse types of applications. The data-center has many shared resources, including the network, and shared resources inside a platform, such as the last-level cache, are prevalent in modern multi-core platforms:

While these shared resources enable high performance scalability and throughput, some programs, such as background video streaming or trans-coding apps, might overuse the cache, decreasing the performance of other critical apps.

3.3.2 How Does Cache Allocation Technology Work

Cache Allocation Technology is a way of splitting the L3 cache into parts and separating them from each other. Specifically Cache Allocation Technology allows us to define classes of service(CLOS) which afterwards we can map cores to CLOS and in this way we achieve cache partitioning. CLOS can also share cache ways between themselves. The above-mentioned interfaces for CLOS deffinition are implemented using Model-Specific-Registries(MSRs).

Each CLOS has a mask which defines the cache ways that it contains, also this cache ways have to be consecutive so for example we can use the 1st and 2nd cache way in a CLOS but we can't use the 1st and the 3rd without the 2nd.

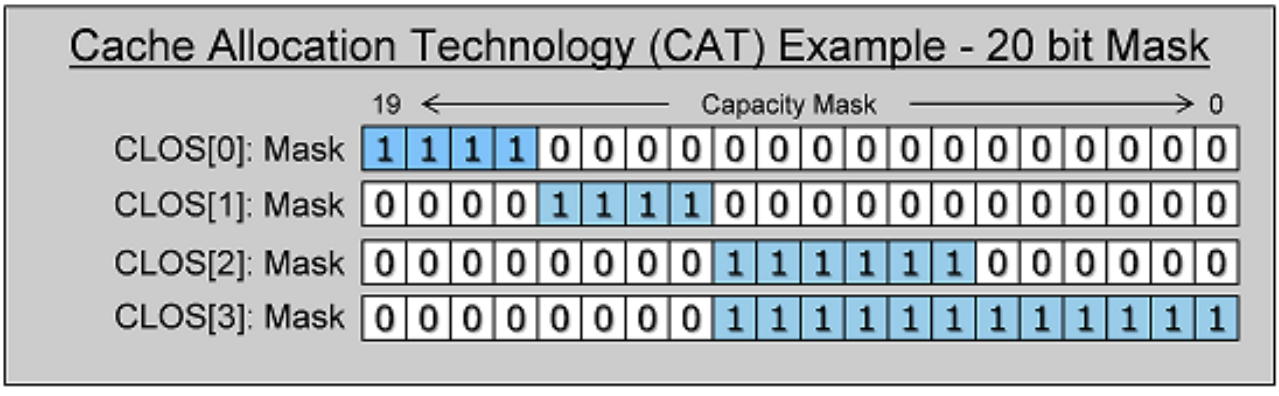

Figure 3.1: Example of Capacity Bitmask (CBM) overlap and isolation across multiple Classes of Service (CLOS)

For example in figure 3.2 we can see the bitmasks of 4 different CLOS, each of this CLOS could have it's own set of cores assigned to it so in practice each core would have a specific set of cache ways to use.

Given software support, Cache Allocation Technology (CAT) allows privileged software like an Operating System or Virtual Machine Manager to manage data placement in the last-level cache (LLC), allowing isolation and priority of key threads, applications, containers, or Virtual Machines. While an early version of CAT was only accessible on a small number of Intel Xeon processor E5-2600 v3 family communications processors, the functionality has been greatly improved and is now available on all SKUs beginning with the Intel Xeon processor E5 v4 family.

3.4 Limiting Process Cores

3.4.1 psutil

Psutil (python system and process utilities) is a cross-platform library for retrieving information on running processes and system utilization (CPU, memory, disks, network, sensors) in Python. It is useful mainly for system monitoring, profiling, limiting process resources and the management of running processes. It implements many functionalities offered by UNIX command line tools such as: ps, top, lsof, netstat, ifconfig, who, df, kill, free, nice, ionice, iostat, iotop, uptime, pidof, tty, taskset, pmap. psutil. In the Linux implementation psutil uses taskset for limiting process cores.

Taskset is used to set or retrieve the CPU affinity of a running process given its pid, or to launch a new command with a given CPU affinity. CPU affinity is a scheduler property that "bonds" a process to a given set of CPUs on the system. The Linux scheduler will honor the given CPU affinity and the process will not run on any other CPUs. Note that the Linux scheduler also supports natural CPU affinity: the scheduler attempts to keep processes on the same CPU as long as practical for performance reasons. Therefore, forcing a specific CPU affinity is useful only in certain applications.

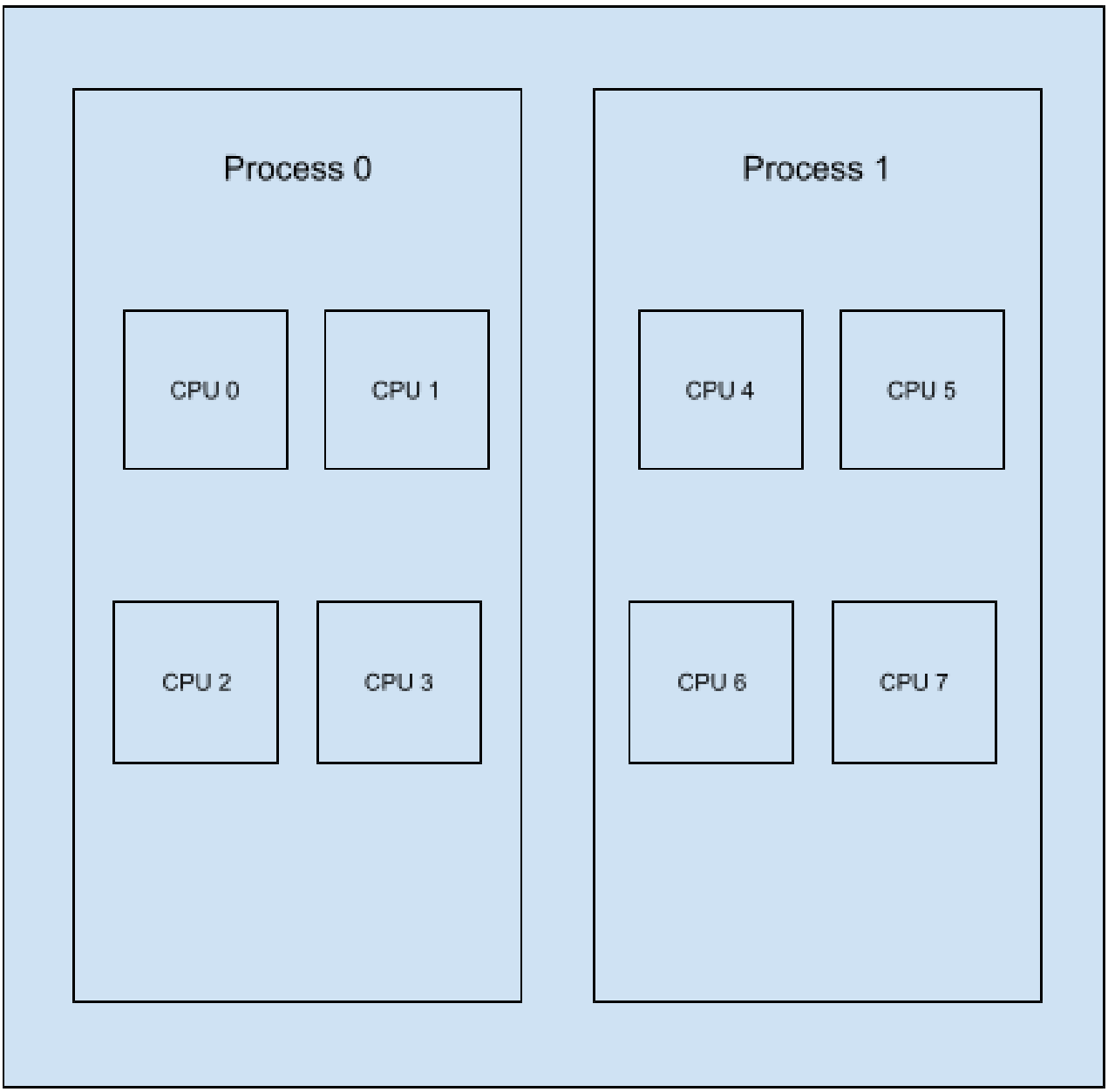

Figure 3.2: Example of a core partitioning

Chapter 4

Experimental Setup & Workload Characterization

In this chapter we evaluate the behavior of single core application in regards to changes in cache memory and the behavior of multi threaded cloud application in regards to increased load so as to find the knee of the application.

4.1 System Setup

Table 4.1: Platform Specification

Model	Intel(R) Xeon(R) CPU E5-2658A v3 @ 2.20GHz
OS	Ubuntu 18.04 (kernel 4.15)
Sockets	2
Cores/Socket	12
Threads/Core	2
Base/Max Turbo Frequency	2.2GHz / 2.9GHz
L1 Inst/Data Cache	32 / 32 KB
L2 Cache	256 KB
L3 (Last-Level) Cache	31 MB
Memory	16GBx8, 2400MHz DDR4

For the rest of the thesis we will work with the system in Table 4.1.

4.2 Characterizing single core applications behavior to changes in cache memory

In this section we evaluate the behavior of single core applications from the SPEC[12] benchmark. The applications aforementioned are gcc, GemsFDTD, leslie3d, libquantum, soplex, sphinx, xalancbmk, zeusmp.

From here on out we will be calling cache sensitive application applications which have a significant impact on performance in regards to the amount of cache ways given to it.

Table 4.2: Single Core Application Characterization

Application Name	IPC (2 ways)	IPC (20 ways)	Percentage Difference
gcc	0.95	1.23	29.47
soplex	1.02	1.12	9.8
sphinx	1.55	2.05	32.26
xalancbmk	1.15	1.65	43.48

As we can see from Figure 4.1 gcc, soplex, sphinx and xalancbmk are all cache sensitive applications as we can see an increase of IPC in the range of 10 to 45 percent from 2 cache ways to 20 cache ways depending on the application.

4.3 Characterizing multi threaded cloud applications behavior in regards to cpu cores

In this section we evaluate the behavior of multi threaded cloud application from the Tailbench[13] benchmarking suite. The applications aforementioned are sphinx, img-dnn, xapian, masstree.

Sphinx[14] is an accurate speech recognition system written in C++. Speech recognition systems are an important component of speech-based interfaces and applications such as Apple Siri,Google Now, and IBM Speech to Text. Speech recognition is a compute-intensive activity, involving probabilistically pruning a large search tree. Sphinx uses sophisticated acoustic, phonetic, and language models to improve efficiency and accuracy. Sphinx is driven using randomly-chosen utterances from the CMUAN4 alphanumeric database.

Img-dnn[15] is a handwriting recognition application based on OpenCV. Handwriting recognition is an example of the broader class of image recognition applications, widely used today for optical character recognition, image-based search(e.g., Google Goggles), automatic image tagging, and a variety of other online applications.Img-dnn uses a deep neural network-based auto encoder coupled with soft max regression to identify handwritten characters. We drive the application using randomly-chosen samples from the MNIST database.

Xapian[16] is an open-source search engine written in C++and widely used both in popular websites (e.g., the Debian wiki) and software frameworks (e.g., Catalyst). Online search engines handle petabytes of index data, which is split into shards spread across thousands of leaf nodes. The bulk of the processing happens at the leaf nodes, with each node independently searching its portion of the index. Xapian is configured to represent a leaf node. The search index is built from a dump of the English version of Wikipedia from July 2013. Query terms are chosen randomly, following a Zipfian distribution, which has been shown to model online search query distributions well.

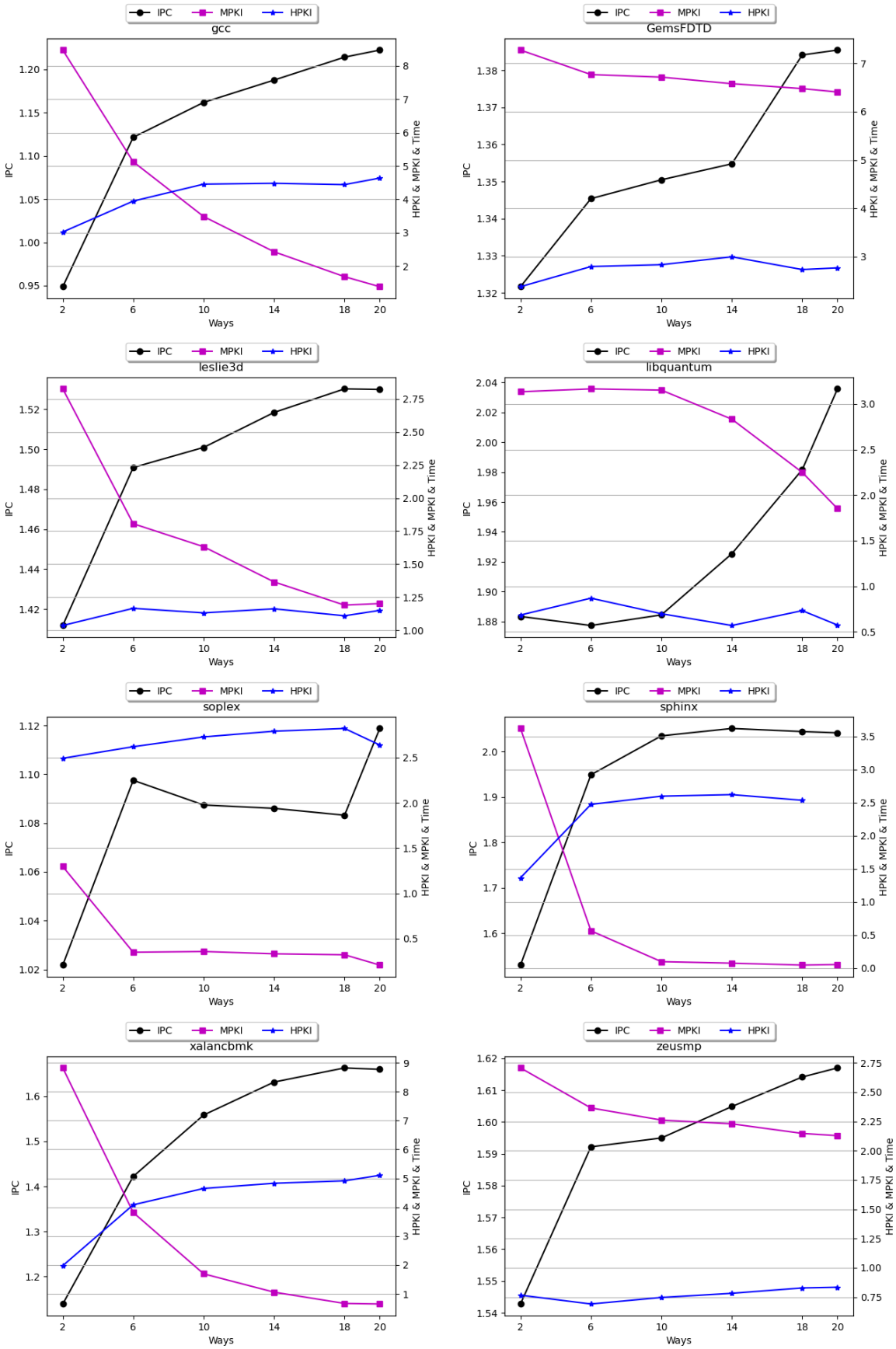

Figure 4.1: (a) gcc (b) GemsFDTD (c) Leslie3d (d) libquantum (e) soplex (f) sphinx (g) xalancbmk (h) zeusmp

masstree[17] is a fast, scalable in-memory key-value store written in C++. In-memory key-value stores serve as data storage backends for a wide variety of services. Key-value stores handle large amounts of data, which is split up into memory-resident shards spread across hundreds of servers.Each user request often involves many tens or hundreds of requests to the key-value store; these applications therefore have very short latency requirements, e.g., about 100µs.

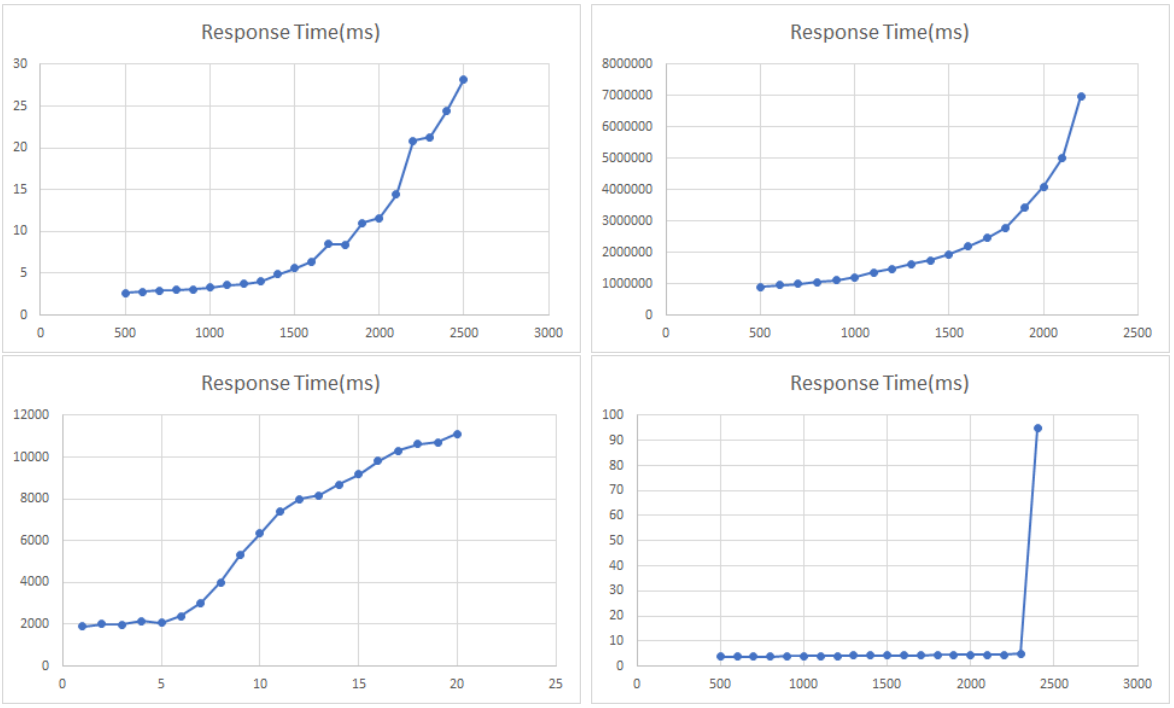

Figure 4.2: (a) img-dnn (b) masstree (c) sphinx (d) xapian
Tail latency with increasing input load (RPS). The vertical lines show the knee of each curve, which is hereafter referred to as maxload. The horizontal lines show the latency at max load, which is used to determine the QoS targets of each application (detailed numbers can be found in Table 4.3).

To find the maximum load the server can maintain and the latency for that load we start from low requests-per-second (RPS) and gradually increase the request rate until we see that the application is forced to drop requests. As we can see for each application there is a load threshold which when exceeded we start seeing exponential growth in the response time of the server. With this in mind we set as the max load of the server for each application the RPS at which we get the 99th percentile(tail) latency of the knee and the response time for that RPS is the Quality-of-Service(QoS) of our application. As we can see in Table 4.3 we have the QoS targets in terms of 99th percentile tail latency and max load so as to maintain QoS target.

Application Name	Max Load under QoS (RPS)	Target QoS (msec)
sphinx	5	2000
masstree	1000	1.2
img-dnn	1300	4
xapian	2300	4.8

Table 4.3: Latency-Critical Applications

4.3.1 Characterization Specifics

For all the latency critical applications above we run the Tailbench benchamrks in integrated mode which means client and application are integrated into a single process and communicate over shared memory so as to nullify any interference from the network since that is not in the scope of this thesis. We assign 24 threads to the application. We warm up each application before starting to aggregate the latencies.

We run the benchmark from a low RPS to a high RPS in steps of 100 (or 1 in the case of sphinx) and we run them for enough time so as to be able to see the impact of passing the threshold, since if we run the benchmark for too little time we may actually lose the real threshold since it will not have as big of an impact if it doesn't have enough time to run.

Chapter 5

Dynamic Resource Tuning of Tail-Latency Applications using DRL

This section introduces our resource manager, a deep reinforcement learning based solution for resource allocation of LC services. The controller leverages a simple DQN architecture that takes PMCs as input and performs fine-grained decisions on the resource allocation. The design goal of the controller is to minimise resource usage while meeting the QoS target of the LC service.

5.1 Design Principles

The resource manager is designed using the following design principles:

- Resource allocation decisions are dynamic and fine-grained. LC applications are very sensitive to resource allocations, with fine-grained resource changes we minimize time lost in thread scheduling and the impact of constant resource allocation changes on cache memory.
- No a priori application knowledge and/or profiling is required.Creating an offline profile of resource interactions in all possible application collocations, even if feasible,would be prohibitively expensive. Moreover, obtaining this information is not always possible, especially in the context of a public cloud hosting previously unknown workloads.
- The controller recovers from incorrect decisions fast.Since the resource manager explores the allocation space online, inevitably some of its decisions may be counterproductive. But the neural network learns fast enough so as to minimize such situations.

5.2 Dynamic Controller Architecture and Specifications

The resource manager consists of:

- System Monitor, which monitors the system and collects the current state of the system as passes it as input to the Deep-Q-Network.
- Deep-Q-Network, which takes as input the current state of the system and calculated the next action.
- Resource Mapper, which takes the decision made by the Deep-Q-Network and does its best to fulfill this decision.

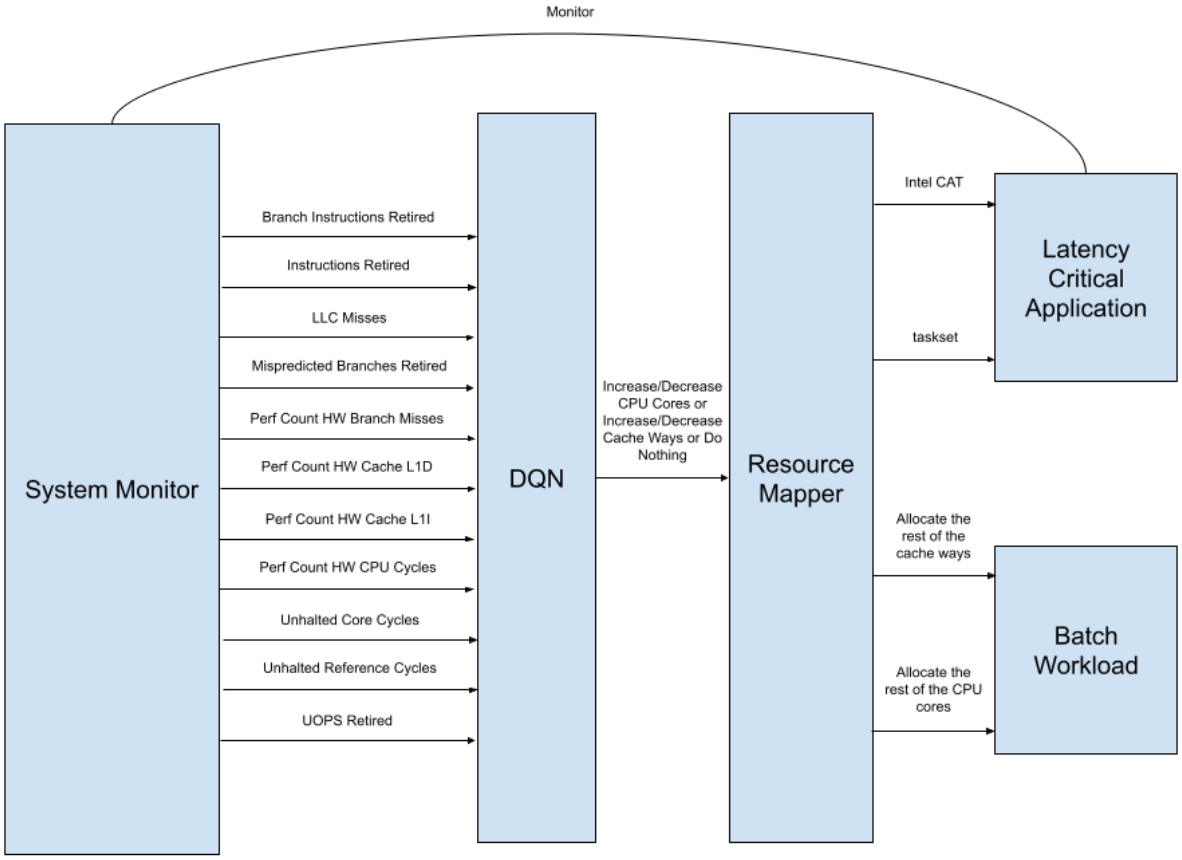

Figure 5.1: An overview of the controller's architecture

5.2.1 System Monitor

The system monitor is responsible for periodically gathering the PMCs at a per-thread level using a profiling tool to measure the activity of each LC service. For each service, we sum the PMCs across all its threads. To reduce the noise over time, a weighted sum for each aggregated counter is computed over the last 5 time steps. The selected PMCs are feature scaled to have values in the range [0,1]. The max-values for the normalisation are collected by running benchmarks that max each one of the PMCs. The data is scaled using max-value normalisation with non-zero centralisation. Feature scaling enables the

neural network to capture the importance of each state variable equally. This data afterwards is fed as the current state to the neural network.

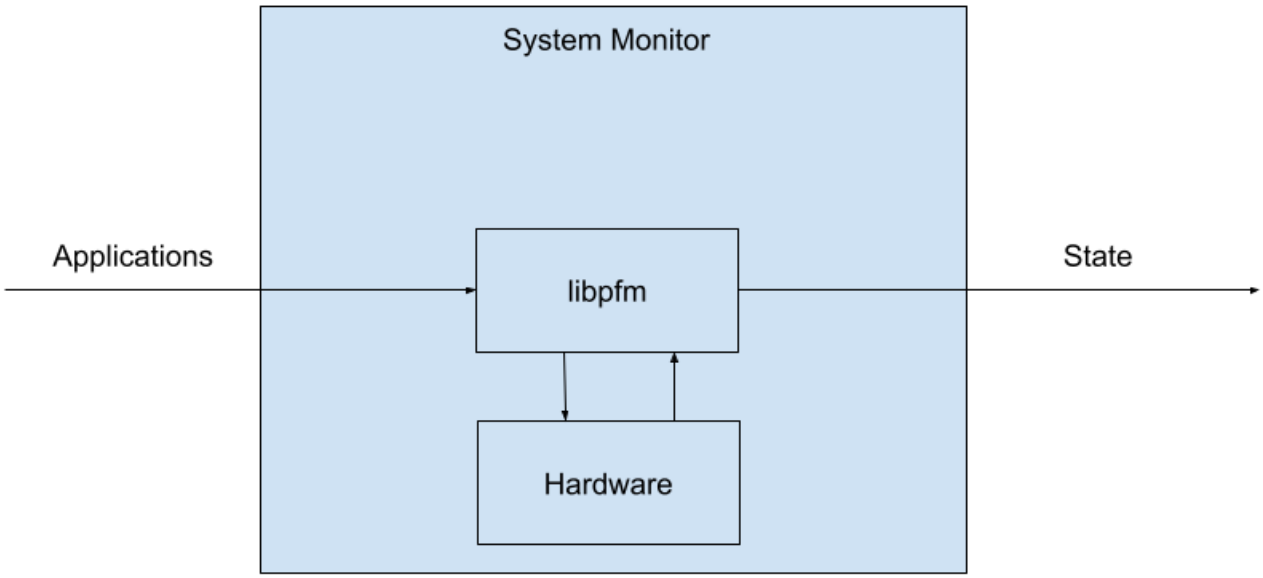

Figure 5.2: A basic overview of the system monitor

5.2.2 Deep-Q-Network(DQN) Agent

The DQN agent learns the "optimal" decisions over time by interacting with the environment using the exploration–exploitation dilemma. In the exploration– exploitation dilemma, the agent not only captures the need to exploit the "optimal" solution found so far but also explores actions that may or may not be better. The probability to explore rather than exploit is captured by epsilon ε. While having a fixed yet small ε is the dominant approach in pure RL settings, it becomes infeasible in large action spaces. The resource manager instead uses epsilon annealing, which transitions from an exploratory policy to an exploitative policy over time for efficient exploration of the discrete action domain. The agent's interactions with the environment at each time step are driven by gathering the state variables, generating actions that are either deterministic or random. The agent then receives a reward in the following time step, determines how well the agent did in the previous time step. The resource manager solves the task management problem by translating it to a Markov decision problem (MDP) that is then solved by a DQN. The DQN Agent is implemented using Stable Baselines 2 and all the application's environment was encapsulated in a OpenAI Gym environment so as to make it as clean and modular as possible. This OpenAI Gym environment then is given to the DQN Agent to train on. The gym environment is responsible for running and monitoring the application, collecting the current state and normalizing it, taking the action that the DQN Agent gives it, perform said action in the best manner

possible and calculating the reward and next state of said action.

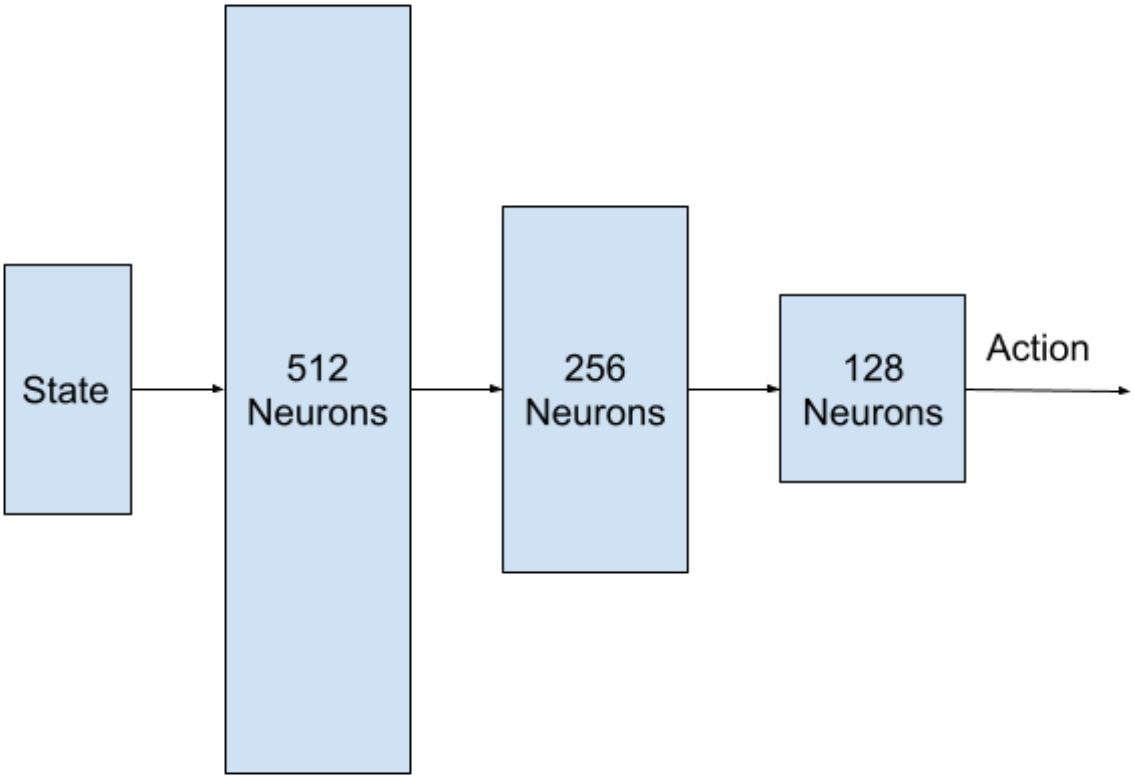

Figure 5.3: A basic overview of the neural networks architecture

Reward Function

The resource manager's reward mechanism determines the mapping decisions based on PMCs and is invoked periodically at each monitoring interval. The reward aims at minimising the resource usage subject to meeting the QoS target, and is expressed as follows:

$$reward = \begin{cases} \frac{QoS_{target}}{QoS} + \frac{20}{currentCacheWays} + \frac{24}{currentCores} & , QoS \leq QoS_{target} \\ max(-(QoS/QoS_{target})^3, -\phi) & , QoS > QoS_{target} \end{cases} \quad (5.1)$$

QoS Reward

If the ratio of the measured QoS to the QoS_{target} is less than or equal to 1, then the QoS target has been met, and quantifies how quick the response was. If this value is greater than 1, then the QoS target has been violated, and therefore we severely penalise the learning agent. As a precursor to ensure that the negative reward is bounded, we cap it to a prefixed value $-\phi$. This value has been found empirically, we've tried many values in the range of (-10,-100). The remarks we made were that for very small values of ϕ the neural network would be more open to violating the QoS constraints, on the other side if the value of ϕ was too large than the neural network would get taxed very heavily for trying to explore at the start of the training phase and as a result it would play it safer by being too generous with the amount of resources it would give to the application. We ended up with a value of ϕ equal to 50 for the purposes of this thesis. Other reward functions we've tried invovled reward functions with the reward variables being exponential bases or powers and doing cross multiplication between them but we found this reward functions being way too unstable for the purpose of this thesis. Intuitively, the only reward that the learning agent should receive is the resource reward (if QoS is met) and a large negative value (if QoS is not met). The part of the reward related to QoS is a heuristic to encourage the algorithm to choose configurations that just meet QoS, which are likely to minimise resource usage (if QoS is met) and tries to reduce the latency in finding an acceptable solution (if QoS is not met).

Resource Reward

The resource reward is given by the ratio of the maximum possible resource allocation to the current resource allocation of the particular resource. A larger value for this term implies that the service's resource usage is lower, and a higher value is added for resource saving.

5.2.3 Resource Mapper

The mapper has three key roles:

- Receive resource allocation requests from the agent
- Ensure the application's threads are mapped to the cores and cache ways allocated.
- Assign any unused resources to the batch workloads

This is done by using Linux's taskset through the python psutil api for pinning threads to cores and Intel CAT's pqos command for assigning cache

ways to the CLOS that the application is running on. If we're running multiple applications in the same time then the policy that the controller uses if first come first served.

Python Psutil

As far as our usage of taskset through the psutil API goes, at first when we launch the latency critical application we keep the parent threads pid and the pids of all the other threads, whenever we want to increase/decrease the cores of a latency critical application we run `psutil.Process(pid=$(PID))` to create a psutil process object and then we use the cpu_affinity method of that object to assign cores to the pid. We do this for the pid of each thread of the latency critical application.

Intel CAT

The way we used Intel CAT was to assign each latency critical applications cores to it's own CLOS using the command `pqos -a "llc:$(CLOS)=$(Cores List);"`. Whenever we increased/decreased cache ways we used the command `pqos -e "llc:$(CLOS)=$(Cache Way Mapping);"` and whenever we increased/decreased cpu cores we used `pqos -a` again with the updated cores list.

5.2.4 Resource Manager Implementation

Table 5.1: Performance Monitoring Counters

#	PMC
1	UNHALTED CORE CYCLES
2	INSTRUCTION RETIRED
3	PERF COUNT HW CPU CYCLES
4	UNHALTED REFERENCE CYCLES
5	UOPS RETIRED
6	BRANCH INSTRUCTIONS RETIRED
7	MISPREDICTED BRANCH RETIRED
8	PERF COUNT HW BRANCH˙MISSES
9	LLC MISSES
10	PERF COUNT HW CACHE L1D
11	PERF COUNT HW CACHE L1I

The resource manager is implemented in user space, and it only uses hardware support exposed by the Linux kernel. The hardware dependent components are PMCs.

PMCs Measurement. The PMCs are measured using the performance monitoring tool perfmon (libpfm 4.10.0). Table 5.1 shows the PMCs that were

selected to input into the DQN agent. The maximum value for counters 1–5 was obtained by running a CPU-intensive microbenchmark consisting of several mathematical operations without memory accesses; counters 6–8 was obtained by running a microbenchmark that generates numerous branch misses by aggregating elements from an unsorted vector of data to check if they are greater than a certain value; counters 9–11 were obtained by running the stream benchmark.

QoS Measurements. As a proof-of-concept, the LC response time is measured by capturing the output from the LC service which outputs it's response time, at a fixed polling interval. An alternative would be to gather the end-to-end latency via the network interface card (NIC), and compute the latency distribution. In a production system, end-to-end latency can be obtained by taking advantage of features of advanced NICs and low-latency networking stacks.

Mapper Module. The services are mapped to cores using the Linux sched setaffinity system call and the cores are mapped to cache ways using Intel CAT.

Neural Network Parameters. We determine through experimental analysis that the following hyper-parameters have yielded the best resource allocation while improving the QoS guarantee. We use the Adam optimise with a learning rate of 0.0025. We set the minibatch size to 64 and the discount factor to 0.99. The target network was updated every 150 time steps. The epsilon annealing starts at 1 and drops to 0.01. We used the rectified non-linearity (ReLU) for all hidden layers and linear activation for output layers. The network has three hidden layers 512,256 and 128. We used the prioritised experience replay with a buffer size of 10^6 and $pr_\alpha = 0.6$, and linear annealing of $pr_\beta = 0.4$ to 1 over 10^{-8} steps and $\phi = 50$.

Implementeion Details. The setup for the evaluation consists of the resource manager and the benchmarking tool. The whole setup is integrated in the OpenAI Gym environment. We achieve this by creating a custom Python class that inherits the base Env class from the gym library. In doing this we must also override the constructor, the step and the reset function.

In the constructor we must first initialize the neural network specific variables provided by the parent class, those are self.action_space and self.observation_space and we initialize them with gym.spaces.Discrete(5)(this means our action space is a discrete number with 5 different actions which are increase/decrease cores, increase/decrease ways, do nothing) and gym.spaces.Box(low=0,high=1.5,shape(13,), dtype=np.float64) (this means our state space consists of 13 floats in the range of 0-1.5) respectively. We also initialize all our inner variables as well as loggers(internal event loggers), cores, core allocation, ways, way allocation and we also start the latency critical application in another thread and keep a pipe to it's standard output so as to be able to get the response times of each request.

The step function takes the current action given by the neural network as

input and in it we define what we do in each cycle of our resource manager and neural network. This includes the sampling the current values of our PMCs(in absolute value), passing the current action to the resource mapper which depending on the action will either increase/decrease cores using taskset or increase/decrease ways using pqos or do nothing. After executing the current action the resource manager sleeps for 2 seconds so as to be able to see the results of it's action, afterwards it recollects the PMCs(in absolute value again) and by having the current PMC values and the previous PMC values we calculate the current state by getting the difference between the two and normalizing the PMC values by their max values(so as to have a value between 0 and 1 which makes it easier for the neural network to train) as well as over the last 5 timesteps(so as to cancel out noise). Also we calculate the reward that the neural network should receive for the last 2 seconds by passing the response times through the reward function. Lastly the step function returns the current state and the reward (which afterwards is passed to Stable Baselines 2 so it can train upon it).

After defining the whole environment we simply initialize the environment class and we also initialize the neural network model using Stable Baselines 2 framework with the hyperparameters defined in the hyper parameters section of 5.2.4 and we also pass the environment as input to the framework.

Chapter 6

Experimental Evaluation

In this chapter, we talk about the benchmarks used in our approach, we talk about the Pearson Correlation of different variables and what that means for out study, we evaluate the results of our proposed approach to different benchmarks as well as how our proposed approach fares in latency critical application collocation with and without batch-workloads. We also calculate the overhead of running our proposed approach and calculate the gain.

6.1 Examined tail-latency applications

For the purpose of this study we needed a benchmarking tool that offered most of the most used cloud applications and has the ability to change the requests per second dynamically in the runtime. To achieve this we used Tailbench[1] with some minor custom modifications so that it reads the RPS over time from a file since Tailbench[1] normally runs with constant RPS. The file contains tuples of RPS and time, it reads the file line by line and executes the RPS from each line for the amount of time given near it in seconds. For the purpose of this study we used the RPS that we found in Table 4.3 as the max load of each application. With this in mind we configured the RPS to slowly increase to the max load(in steps of 100) and then decrease again so the neural network can train on all of the applications states as seen in figure 6.1.

The benchmarks used in the experiments below are img-dnn and masstree.

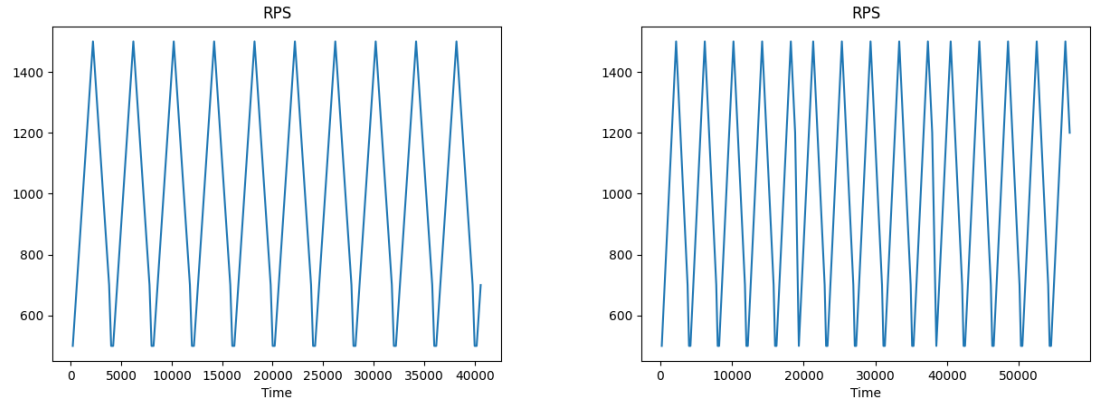

Figure 6.1: (a) img-dnn (b) masstree

6.2 Performance Monitoring Counters (PMCs)

In this section we will establish the relationship between the PMCs and requests-per-second(RPS).

By running img-dnn while collecting it's PMCs we get Figure 6.2.

And by running masstree while collecting it's PMCs we get Figure 6.3.

As we can see the PMCs follow really closely the RPS wave and by just using the PMCs we can deduce the RPS of the application.

With this in mind we can see how the PMCs are a good input for the neural network to be able to deduce the state of the application.

6.2.1 Pearson Correlation Between PMCs and RPS

In this subsection we will establish the Pearson Correlation between each PMCs and RPS.

In statistics, the Pearson correlation coefficient is a measure of linear correlation between two sets of data. The reason we would like to find the Pearson Correlation between the PMCs and the RPS is because in this section we would like to prove that by having the PMCs one can actually deduct the current RPS (since in a real life dynamic system we can only get the RPS for the previous time-step). By showing the one can deduct the current RPS from the PMCs we also show that the neural network itself can deduct the RPS from the PMCs so the resource management problem translated into a RPS to action problem, which is a lot simpler.

We calculate the Pearson Correlation by using the formula below:

$$r = \frac{\sum_i (x_i - \tilde{x})(y_i - \tilde{y})}{\sqrt{\sum_i (x_i - \tilde{x})^2 \sum_i (y_i - \tilde{y})^2}} \tag{6.1}$$

Where x is the RPS and y is the PMCs.

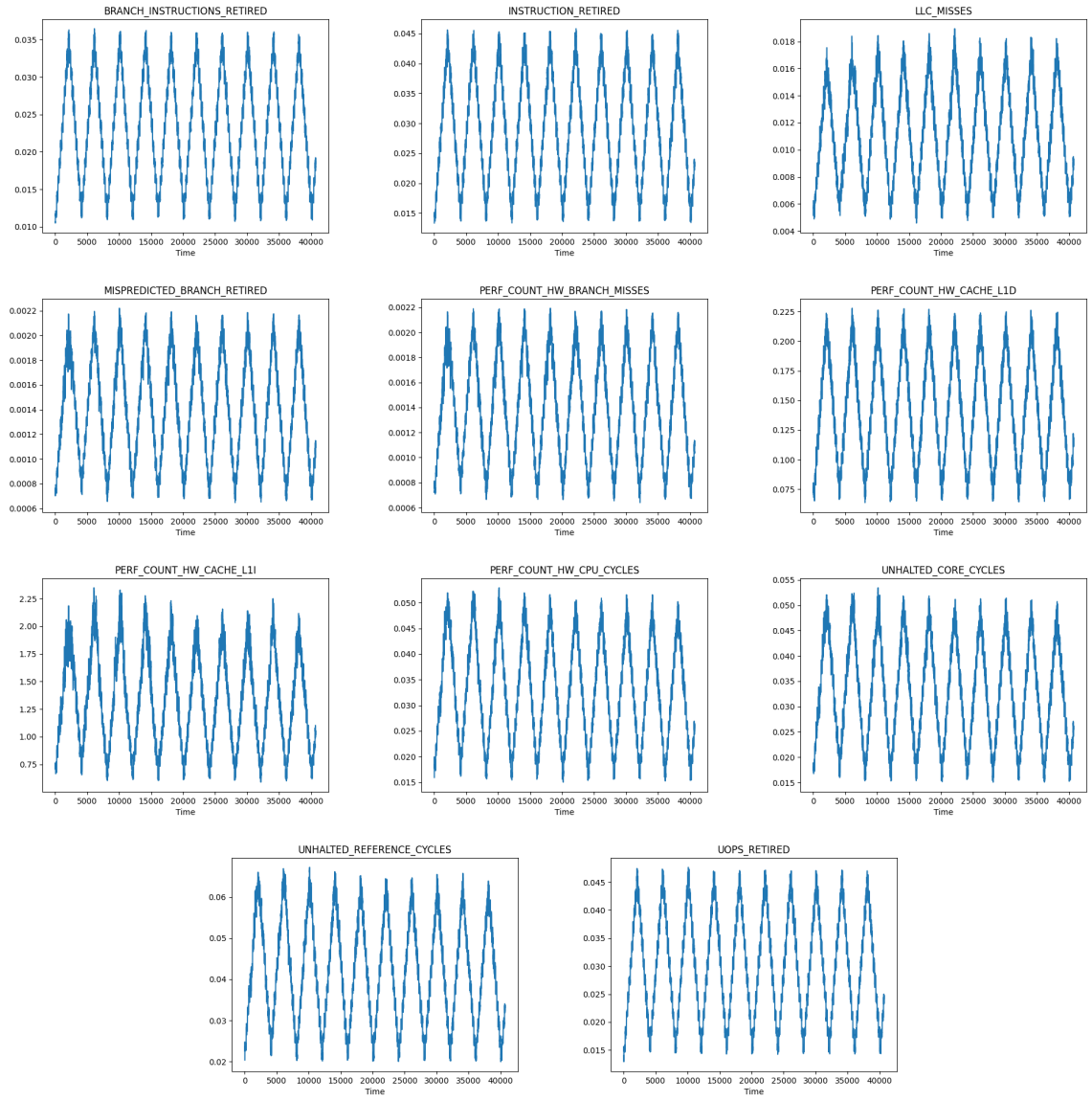

Figure 6.2: Performance Monitoring Counters

The value of r is in the range of $[-1, -1]$ and symbolizes how correlated the two variables are. If r is equal to 1 then it means that the two variables are strongly correlated and their values behave in the same way over time. If r is equal to -1 then it means the the two variables are again strongly correlated but their values behave in the same way but in opposite directions.

In tables 6.1 and 6.2 we can see the Pearson Correlation between the PMCs and the RPS for img-dnn and for masstree. We can obviously see that the Pearson Correlation is very strong (as anticipated by looking at the RPS and PMC graphs)

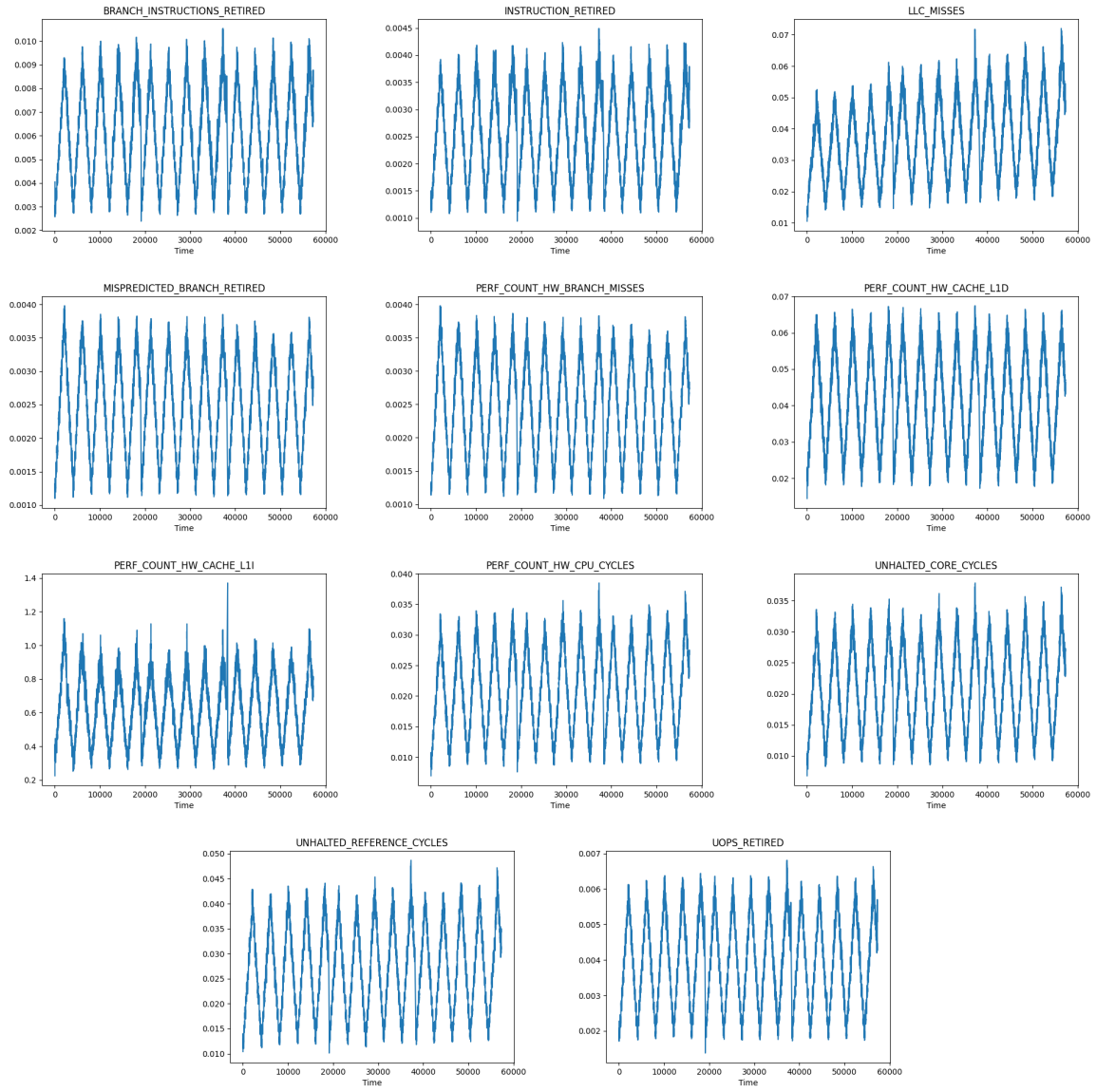

Figure 6.3: Masstree Performance Monitoring Counters

6.3 Performance evaluation without interference

6.3.1 img-dnn

In figure 6.4 we can see the first 10.000 seconds of training for img-dnn. As we can see the first 10.000 seconds are very chaotic as the neural network tries all kinds of state-action combinations so it can learn. Afterwards in figure 6.5 we can see how the resource manager behaves once it's fully trained. As we can see the number of cores and ways allocated end up following the RPS in behavior as the neural network trains. Also the response time is almost always under the QoS target even in the worst case scenario of training which is at the 8000 seconds.

Also we can see that by leveraging a good resource allocation the controller can achieve high QoS quarantee while using only 2 to 8 cores and 2 to 8 cache

Table 6.1: Pearson Correlation Between PMCs and RPS for img-dnn

PMC	Pearson Correlation
BRANCH INSTRUCTIONS RETIRED	0.99
INSTRUCTIONS RETIRED	0.99
LLC MISSES	0.99
MISSPREDICTED BRANCH RETIRED	0.99
PERF COUNT HW BRANCH MISSES	0.99
PERF COUNT HW CACHE L1D	0.99
PERF COUNT HW CACHE L1I	0.99
PERF COUNT HW CPU CYCLES	0.99
UNHALTED CORE CYCLES	0.99
UNHALTED REFERENCE CYCLES	0.99
UOPS RETIRED	0.99

Table 6.2: Pearson Correlation Between PMCs and RPS for masstree

PMC	Pearson Correlation
BRANCH INSTRUCTIONS RETIRED	0.97
INSTRUCTIONS RETIRED	0.98
LLC MISSES	0.90
MISSPREDICTED BRANCH RETIRED	0.97
PERF COUNT HW BRANCH MISSES	0.99
PERF COUNT HW CACHE L1D	0.99
PERF COUNT HW CACHE L1I	0.96
PERF COUNT HW CPU CYCLES	0.98
UNHALTED CORE CYCLES	0.98
UNHALTED REFERENCE CYCLES	0.98
UOPS RETIRED	0.99

ways

In figure 6.6(a) we can also see that the QoS quarantee as the application is running is in the worst case scenario(which is at the start of the training phase) above 92%

But since we care most for the QoS quarantee of the trained model if we look at the QoS quarantee only for data after 30000 seconds we get the QoS quarantee in figure 6.6(b).

Resource Gain

During the img-dnn run above the average response time for the trained neural network was 3,21 millisecond. To achieve this kind of average response time while maintaining a QoS quarantee over 99% over the whole duration of a cycle without using the resource manager and by just setting the latency critical application on a fixed number of resources we would need 7 CPU cores and 5 cache ways. By calculating the average CPU cores and cache ways during the

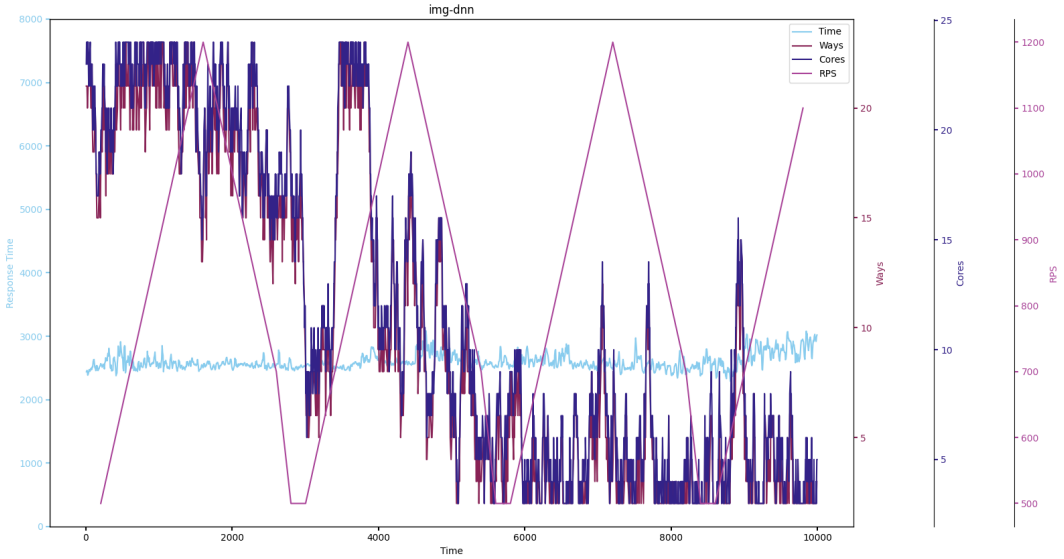

Figure 6.4: img-dnn during the first 10.000 seconds of training

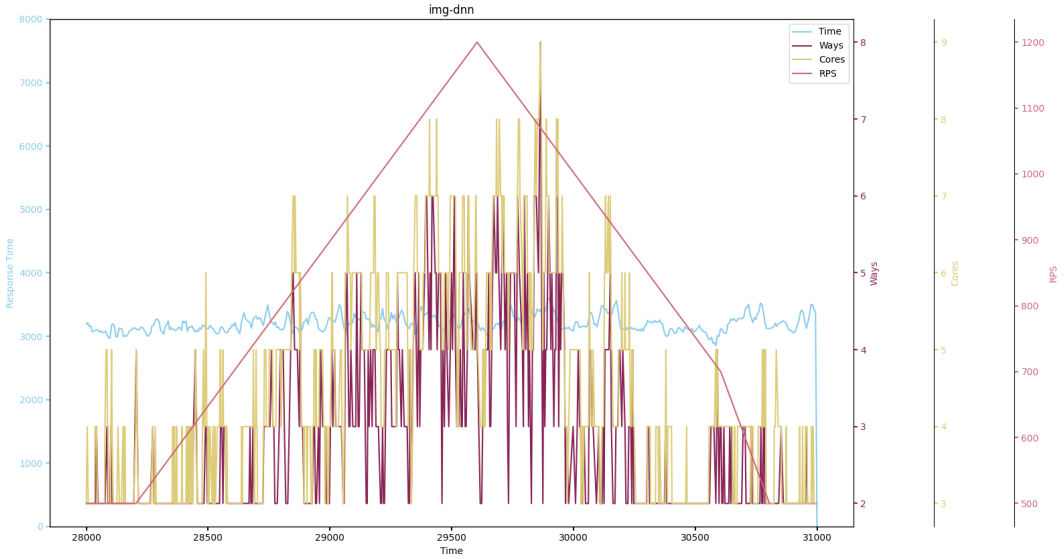

Figure 6.5: img-dnn once it has trained

once cycle in figure 6.5 we got 4.39 CPU cores and 3.76 cache ways. As we can see we have a 37% decrease in average CPU usage and a 24% decrease in average cache way usage.

6.3.2 masstree

In figure 6.7 we can see the first 10.000 seconds of training masstree, as before with img-dnn the first training phase is very chaotic since the neural network is trying to explore the state-action space and learn. Afterwards in figure 6.8

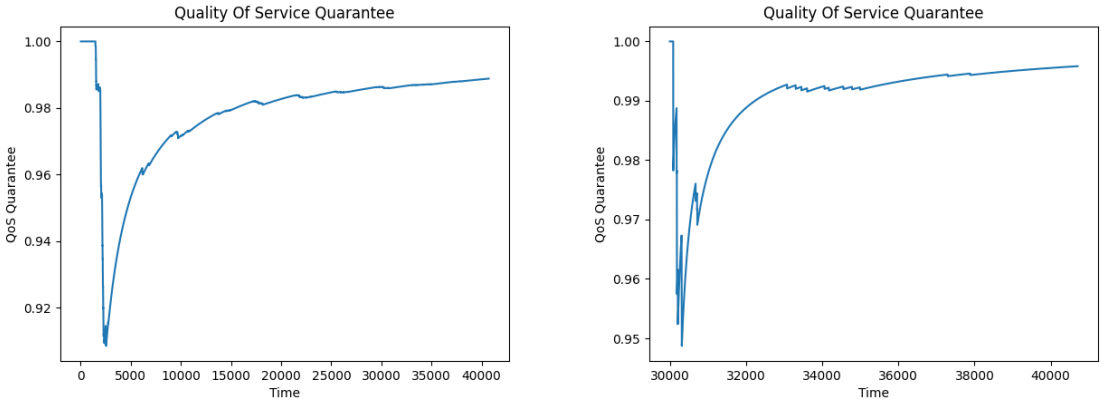

Figure 6.6: (a) QoS quarantee img-dnn (b) QoS quarantee trained img-dnn

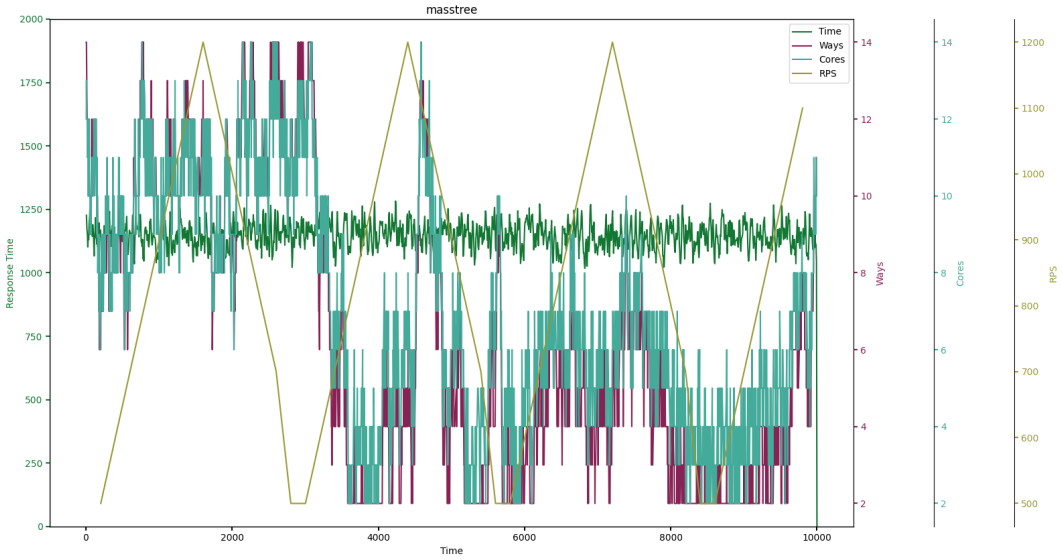

Figure 6.7: Masstree during the first 10.000 seconds of training

we can see how the resource manager behaves when it's fully trained. Also we can see the number of cores and ways allocated end up following the RPS in behavior as the neural network trains. Also the response time is almost always under the QoS target.

Again we can see that by leveraging a good resource allocation the controller can achieve high QoS quarantee while using only 2 to 10 cores and 2 to 9 cache ways

In figure 6.9(b) we can also see the QoS quarantee after the training phase is around 99%.

Resource Gain

During the masstree run above the average response time for the trained neural network was 1.12 millisecond. To achieve this kind of average response time

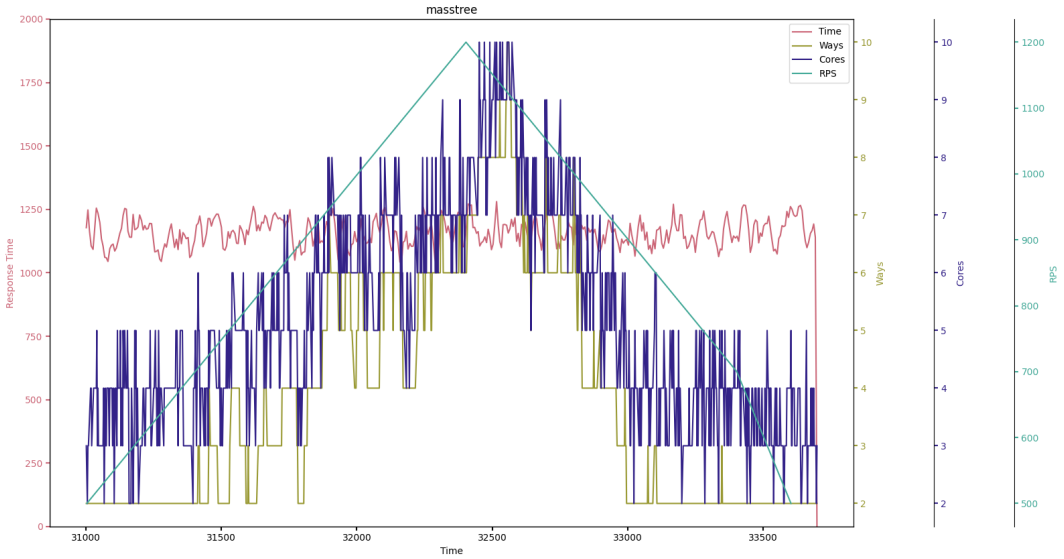

Figure 6.8: Masstree once it has trained

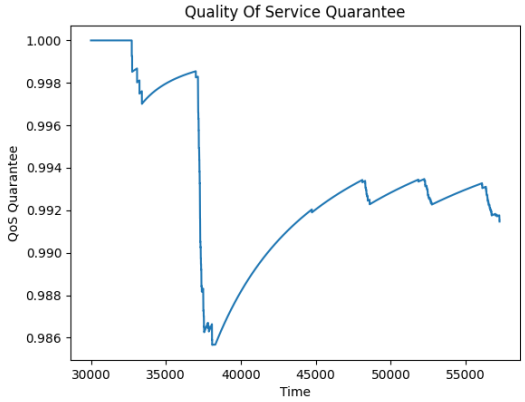

Figure 6.9: img-dnn QoS Quarantee on a trained model

while maintaining a QoS quarantee over 99% over the whole duration of a cycle without using the resource manager and by just setting the latency critical application on a fixed number of resources we would need 8 CPU cores and 8 cache ways. By calculating the average CPU cores and cache ways during the once cycle in figure 6.8 we got 5.23 CPU cores and 4.12 cache ways. As we can see we have a 34% decrease in average CPU usage and a 48% decrease in average cache way usage.

6.4 Comparison of Deep-Q Learning with other Deep Reinforcement Learning Algorithms

In this section we will analyse the reasons why we choose a Deep-Q-Network for this thesis over other reinforcement learning algorithms.

First of all when choosing a reinforcement learning algorithm we must think about our state-space and action-space since not all reinforcement learning work on all combinations of state-action space. In our scenario we have a continuous state-space(since our inputs are float values between 0 and 1) and a discrete action-space(since we have 5 different choices of an action). Also we must consider the complexity of each algorithm, the tuning difficulty, sample efficiency(since we wait for each sample 2 seconds) and noise robustness.

With the above mentioned criteria in mind we will discuss other reinforcement learning algorithms and why we can't use them.

Deep Deterministic Policy Gradient (DDPG) is a reinforcement learning technique that combines both Q-learning and Policy gradients. DDPG being an actor-critic technique consists of two models: Actor and Critic. The actor is a policy network that takes the state as input and outputs the exact action (continuous), instead of a probability distribution over actions. The critic is a Q-value network that takes in state and action as input and outputs the Q-value. DDPG is an "off"-policy method. DDPG is used in the continuous action setting and the "deterministic" in DDPG refers to the fact that the actor computes the action directly instead of a probability distribution over actions. As we can see from above DDPG outputs a continuous action(we want a discrete action).

Soft Actor-Critic (SAC) is defined for RL tasks involving continuous actions. The biggest feature of SAC is that it uses a modified RL objective function. Instead of only seeking to maximize the lifetime rewards, SAC seeks to also maximize the entropy of the policy. The term 'entropy' has a rather esoteric definition and many interpretations depending on the application but I'd like to share an intuitive explanation here. We can think of entropy as how unpredictable a random variable is. If a random variable always takes a single value then it has zero entropy because it's not unpredictable at all. If a random variable can be any Real Number with equal probability then it has very high entropy as it is very unpredictable. Again as before SAC is used only for continuous actions.

Asynchronous Advantage Actor-Critic (A3C) is one of RL's state-of-the-art algorithms, which beats DQN in few domains. Also, A3C can be beneficial in experiments that involve some global network optimization with different environments in parallel for generalization purposes. Asynchronous stands for the principal difference of this algorithm from DQN, where a single neural network interacts with a single environment. On the contrary, in this case, we've got a global network with multiple agents having their own set of parameters. It creates every agent's situation interacting with its environment and harvesting the different and unique learning experience for overall training.

That also deals partially with RL sample correlation, a big problem for neural networks, which are optimized under the assumption that input samples are independent of each other (not possible in games). The problem with A3C in our case is that we can't just spin a big number of LC application instances because we will start having bottlenecks and interference between LC application.

Proximal Policy Optimization (PPO) is a policy gradient method and can be used for environments with either discrete or continuous action spaces. It trains a stochastic policy in an on-policy way. Also, it utilizes the actor critic method. The actor maps the observation to an action and the critic gives an expectation of the rewards of the agent for the observation given. Firstly, it collects a set of trajectories for each epoch by sampling from the latest version of the stochastic policy. Then, the rewards-to-go and the advantage estimates are computed in order to update the policy and fit the value function. The policy is updated via a stochastic gradient ascent optimizer, while the value function is fitted via some gradient descent algorithm. This procedure is applied for many epochs until the environment is solved. The other deep reinforcement learning algorithm we considered in this thesis was PPO since it can also work on continuous state-space and discrete action-space.

Proximal Policy Optimization Evaluation

For our evaluation of PPO we tried to train img-dnn using it.

The **hyperparameters** we used for this evaluation are discount factor equal to 0.99, number of steps per update were equal to 128, entropy coefficient equal to 0.01, learning rate equal to 0.0025, value function coefficient equal to 0.5, maximum value for the gradient clipping equal to 0.5, number of training mini-batches per update equal to 4, number of epoch when optimizing the surrogate equal to 4.

We can see in figure 6.10 the first 10 000 seconds of the training phase during which the PPO in comparison to the DQN trains a lot slower and does more radical changes as a result we see really huge spikes in response time.

Also we can see in figure 6.11 PPO is obviously overestimating the amount of CPU cores needed to maintain the QoS(most likely because it's not very flexible to noise). Also we can see that the response time isn't as stable and violates a lot of times the QoS target even though it gives almost all of the available cores, the reason for this is because of the huge changes in cores that it does very quickly as a result the system loses a lot of time in rescheduling threads to cores and it also trashes the cache.

In any other scenario PPO would most likely be a lot better than DQN. But in our thesis because of the nature of our environment, by that we mean little samples, simple neural network, lots of noise(because of the nature of a

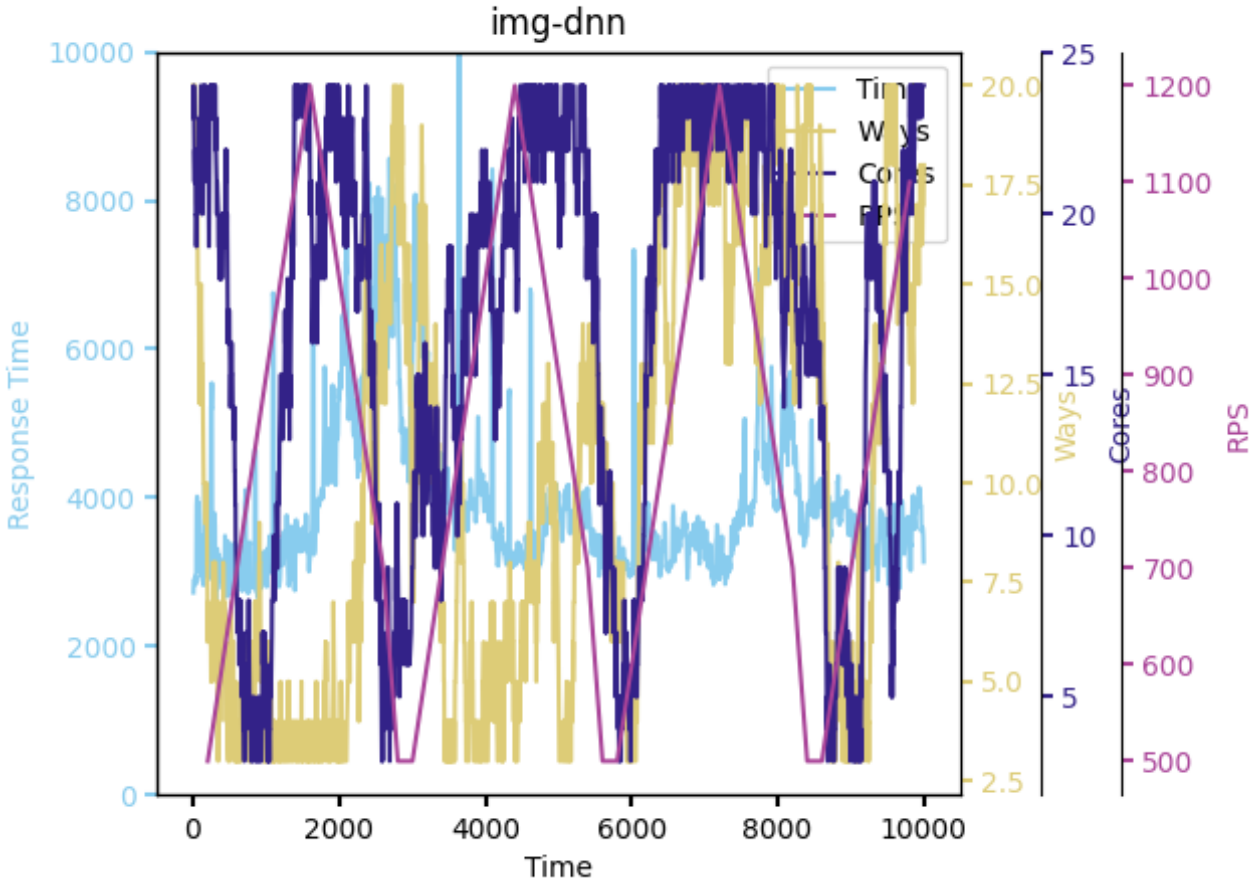

Figure 6.10: img-dnn with PPO during the first 10.000 seconds of training

production server), we need a reinforcement learning that is sample efficient, with a small footprint and and is robust to noise. That's why in the end DQN(with all of it's improvements(this are double q learning and prioritized replay) ends up performing better in our environment.

6.5 Performance evaluation under interference

6.5.1 Concurrent Execution of LC Applications

In figure 6.12 we had a trained img-dnn model and a trained masstree model run in parallel with no interference. As we can see the neural network achieves a stable response time that is within the QoS target. We can also see how the total resource usage is way bellow 100% . For example at the peaks the maximum cores used by img-dnn are 8 and the maximum cores used by masstree is also 8. As the system has a total of 24 cores we see that even at the peak of RPS we still have 8 cores left for batch workloads. With the same logic the number of cache ways used during the peaks are 4 for img-dnn and 4 for masstree and since we have 20 total cache ways we have 12 cache ways left for

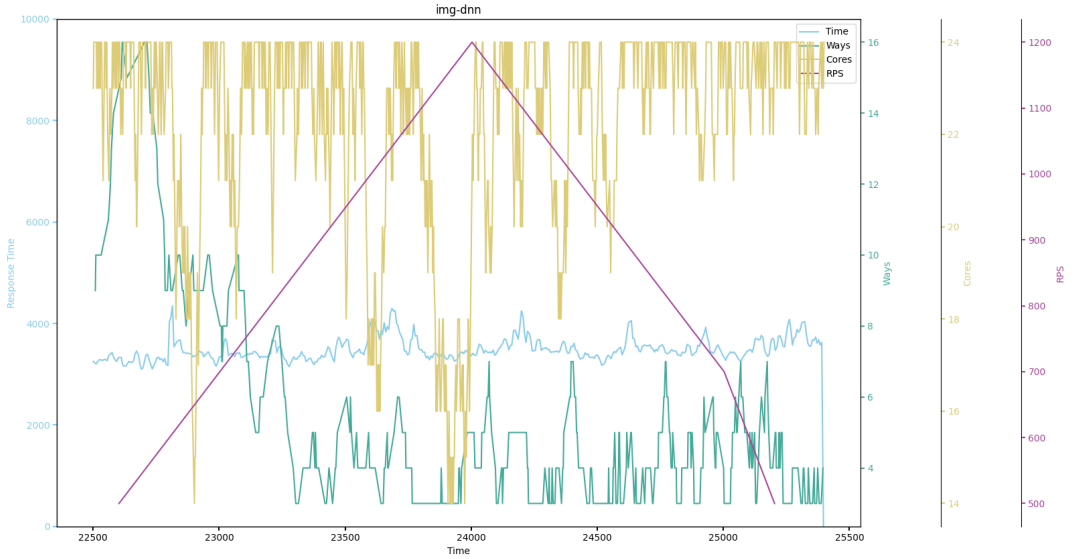

Figure 6.11: img-dnn with PPO once it has trained

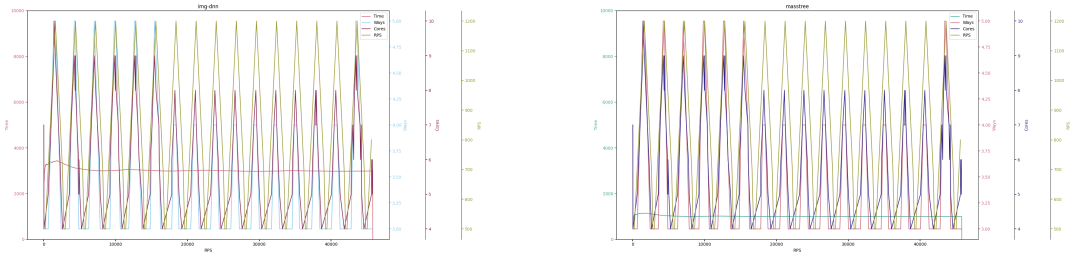

Figure 6.12: (a) img-dnn (b) masstree

batch workloads.

6.5.2 Concurrent Execution of LC and Batch-Workload Applications

In figure 6.13 we had a trained img-dnn model and a trained masstree model run in parallel with interference. The interference in this study is another instance of vanilla Tailbench[1](meaning without our minor changes) running Sphinx with constant requests-per-second(so as to simulate a constant load batch-workload) equal to 3 requests-per-second(the reason it's so low is because each request of Sphinx has around 3-4 seconds response time and it's a very taxing application that uses CPU cores and cache ways really heavily). As we can see the neural network achieves a stable response time that is within the QoS target. We can also see how the total resource usage is way bellow 100% . For example at the peaks the maximum cores used by img-dnn are 9-10 and the maximum cores used by masstree is 8. As the system has a total of 24 cores we see that even

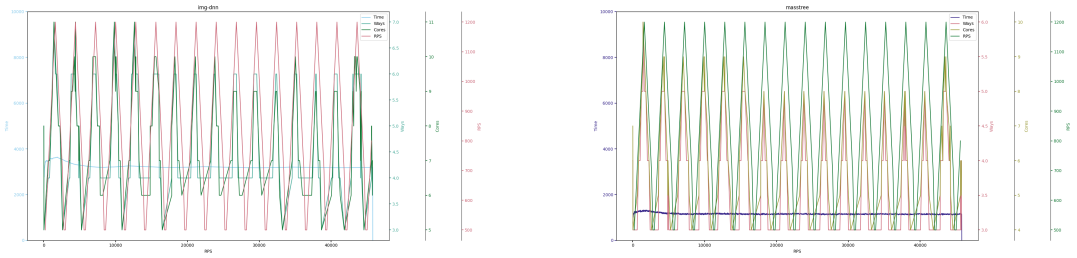

Figure 6.13: (a) img-dnn (b) masstree

at the peak of RPS we still have 6 cores left for batch workloads. With the same logic the number of cache ways used during the peaks are 6 for img-dnn and 5 for masstree and since we have 20 total cache ways we have 9 cache ways left for batch workloads. In this experiment the batch workload was another Tailbench[1] benchmark, specifically it was Tailbench[1] Sphinx.

6.6 Controller's Overhead Over Running Applications

In this section we will be calculating the overhead of the neural network with two different methodologies.

One methodology will be calculating the overhead by running the benchmarks for a fixed number of requests with and without the neural network.

The other methodology includes calculating the time each part of the application takes in run-time.

For the first methodology we will calculate the overhead for 3 different situations. The first is if we have the neural network and the application running on different sockets. The second is if we have the neural network and the application running on the same socket but on different physical cores. The third is if we have the neural network and the application running on the same physical core but on different hardware threads.

6.6.1 Overhead Calculation Using Fixed Requests

We ran the benchmark for 100.000 requests without the neural network and for 100.00 with the neural network. We did this 100 times and calculated the averages.

Different Sockets

By doing this we found that without the neural network the benchmark needs 392 second to finish 100.000 requests and with the neural network the benchmark needs 392. As expected there's no overhead when we have the neural network and the application on different sockets.

Same Socket Different Psychical Cores

By doing this we found that without the neural network the benchmark needs 392 second to finish 100.000 requests and with the neural network the benchmark needs 398.

$$Overhead = \frac{398 - 392}{392} = 0.016 = 1.6\% \tag{6.2}$$

As expected we see a big increase in overhead in comparison to running on different sockets. The reason this is expected is because by running both the neural network and the application on the same socket we keep on changing data in the L3 cache(since a neural network uses cache memory a lot during each pass).

Same Physical Core Different Hardware Threads

By doing this we found that without the neural network the benchmark needs 392 second to finish 100.000 requests and with the neural network the benchmark needs 404.

$$Overhead = \frac{404 - 392}{392} = 0.03 = 3\% \tag{6.3}$$

As expected we see a big increase in overhead in comparison to running on the same socket with different physical cores. The reason this is expected is because by running both the neural network and the application on the same physical core we keep on changing data in the L1 and L2 caches(since a neural network uses cache memory a lot during each pass).

	Different Sockets	**Different Physical Cores**	**Same Physical Core**
With Neural Network	392	398	404
Without Neural Network	392	392	392
Overhead	0%	1.6%	3%

Table 6.3: Overhead for each situation

6.6.2 Overhead Calculation Using Run-time

The second way to find the overhead was to find how much time each part of the neural network takes and since we run the neural network every 2 seconds the overhead would be the extra time needed. This overhead is calculated with the neural network and the application on the same socket but on different physical cores.

$$Overhead = \frac{2.032 - 2}{2} = 0.015 = 1.5\% \tag{6.4}$$

Table 6.4: Overhead

Gradient descent computation	21 ms
Gather and preprocess PMCs	3 ms
Core/Way Allocation	8 ms
Total Overhead	32 ms

As we can see from the first method we found an overhead of 1.6% and from the second method we found an overhead of 1.5% when the neural network and the application are on the same socket but on different physical cores, as we can see both method pretty much agree on the overhead of the neural network.

6.7 Evaluation Summary

With all the evaluations above we came to the conclusion that the resource manager can achieve a big decrease in CPU and cache usage(20 to 40 percent) while maintaining a high QoS guarantee(above 99 percent) and maximising batch workload performance(since all the freed resources will go to the batch workloads) with a small overhead of 1.6%.

Chapter 7

Conclusion

In this thesis, we discussed about the current state of cloud services and how they are used in the modern cloud environment. We also talked about state-of-the-art machine learning techniques and we showcased the performance effect that CPU Cores and Cache Ways can have on the performance of a cloud service.

A big concern of the modern cloud environment is the low resource usage. Attempting to better understand this phenomena, we analyzed the performance of workloads from different benchmarking libraries, under pressure on various resources in different intensities.

Thus, we identified that there are possibilities for better resource usage optimization. Also we identified the need for a benchmarking tool with dynamic requests and did minor changes to a popular benchmarking tool to achieve this.

Furthermore, we designed a deep reinforcement learning based resource manager for cloud services in a simple and modular way that makes it easy to change neural network architecture, reinforcement learning algorithm and hyper parameters while maintaining a high standard of performance. Our resource manager uses complex performance monitoring counters so as to be able to determine the system's state and make better decisions.

We showed that in most of the scenarios our resource manager can achieve almost 100% resource usage while maintaining a 99% quality-of-service guarantee.

Also we showcased that our resource manager works with latency critical application co-location and that our resource manager can even achieve multiple latency critical application co-location while maximising batch-workload performance.

Finally we evaluated our resource manager's overhead and determined it to be 1.6% if we place the resource manager and the latency critical application on the same socket but on different physical cores which is the most likely scenario.

Bibliography

[1] H. Kasture and D. Sanchez, "Tailbench: a benchmark suite and evaluation methodology for latency-critical applications," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*, 2016, pp. 1–10.

[2] B. Copeland, "Artificial intelligence," *Encyclopedia Britannica*.

[3] "Cache memory," *Encyclopedia Britannica*.

[4] A. C. Sodan, J. Machina, A. Deshmeh, and K. Macnaughto, "Parallelism via multithreaded and multicore cpus," *IEEE Xplore*.

[5] S. Chen, C. Delimitrou, and J. Martínez, "Parties: Qos-aware resource partitioning for multiple interactive services," 04 2019, pp. 107–120.

[6] N. El-Sayed, A. Mukkara, P.-A. Tsai, H. Kasture, X. Ma, and D. Sanchez, "Kpart: A hybrid cache partitioning-sharing technique for commodity multicores," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 104–117.

[7] J. Park, S. Park, and W. Baek, "Copart: Coordinated partitioning of last-level cache and memory bandwidth for fairness-aware workload consolidation on commodity servers," in *Proceedings of the Fourteenth EuroSys Conference 2019*, ser. EuroSys '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3302424.3303963

[8] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers," vol. 41, 07 2013, pp. 607–618.

[9] R. Nishtala, V. Petrucci, P. Carpenter, and M. Själander, "Twig: Multi-agent task management for colocated latency-critical cloud services," 12 2019.

[10] T. Ayodele, *Machine Learning Overview*, 02 2010.

[11] S. Mousavi, M. Schukat, and E. Howley, "Deep reinforcement learning: An overview," 06 2018, pp. 426–440.

[12] "Spec," https://www.spec.org/benchmarks.html.

[13] "Tailbench: a benchmark suite and evaluation methodology for latency-critical applications," *IEEE International Symposium on Workload Characterization*, 2016.

[14] P. Lamere, P. Kwok, E. Gouv, R. Singh, W. Walker, and P. Wolf, "The cmu sphinx4 speech recognition system," *IEEE Intl. Conf. on Acoustics, Speech and Signal Processing (ICASSP 2003)*, 01 2003.

[15] "A deep network handwriting classifier." [Online]. Available: https://github.com/xingdi-eric-yuan/multi-layer-convnet

[16] "Xapian project." [Online]. Available: https://github.com/xapian/xapian

[17] Y. Mao, E. Kohler, and R. Morris, "Cache craftiness for fast multicore key-value storage," *EuroSys'12 - Proceedings of the EuroSys 2012 Conference*, 09 2012.